

IDC DOCUMENTATION

Continuous Data Subsystem CD-1.1



Notice

This document was first published in March 2001 by the Monitoring Systems Operation of Science Applications International Corporation (SAIC) as part of the International Data Centre (IDC) Documentation. It was revised to add multicasting capability and republished as Revision 1 in January 2003 (see the following [Change Page](#)). Every effort was made to ensure that the information in this document was accurate at the time of publication. However, information is subject to change.

Contributors

Raymond L. Cordova, Science Applications International Corporation
Aaron Douthat, Science Applications International Corporation
Warren K. Fox, Science Applications International Corporation
Jeffrey W. Given, Science Applications International Corporation
Bonnie MacRitchie, Science Applications International Corporation
Thomas Morris, Science Applications International Corporation
Ellsworth Sacks, Science Applications International Corporation

Trademarks

Ethernet is a registered trademark of Xerox Corporation.
NFS is a registered trademark of Sun Microsystems.
ORACLE is a registered trademark of Oracle Corporation.
Solaris is a registered trademark of Sun Microsystems.
SPARC is a registered trademark of SPARC International, Inc.
Sun is a registered trademark of Sun Microsystems.
UNIX is a registered trademark of UNIX System Labs, Inc.

Ordering Information

The ordering number for this document is SAIC-03/3000.

This document is cited within other IDC documents as [IDC7.4.1Rev1].

Change Page

This document is Revision 1 of the Continuous Data Subsystem CD-1.1. The following changes have been made for this publication:

| Date | Page | Change |
|--------------|-----------------------------|---|
| January 2003 | as needed | Added change bars (vertical lines that appear in the margin) to identify new or revised material. Change bars were not included for references that were updated, for grammar and punctuation that was corrected, or for figures that were moved to accommodate sensible paging. |
| | Notice | Changed Notice Page to include an explanation of the Revision number, a reference to this Change Page, and a new SAIC number. Added Bonnie MacRitchie as an author. Added Trademark notifications for Ethernet, NFS, and SPARC. |
| | Change Page | Added this Change Page. |
| | ii | Added paragraphs describing the changes made since the last version. |
| | 5 | Added the multicast technology and distinguished between unicast and multicast data flows in Figure 2. |
| | 6 | Changed "via the CD-1.1 protocol" to "via unicast or multicast CD-1.1 protocol" in the description of Functionality. |
| | 6 | Changed "Foreman" to "ForeMan" in the Identification bullets. |
| | 6 | Added "Multicast Sender (MCastProvider)", "Multicast Receiver (MCastConsumer)", and "Missing Frame Detector (FrameAudit)" to the Identification bullets. |
| | 7 | Removed the Status of Development section. |
| | 7 | Added "In 2002 multicast capability was added to CDS CD-1.1." to the Background and History section. |
| | 7, 8-9 | Added a "Network Environment" section to Operating Environment. |
| | 7 | Deleted reference to existing implementations in Hardware paragraph. |
| | 11, 34-42 | Added a "Software Components" section. |

| Date | Page | Change |
|--------------|---|--|
| January 2003 | 13 | Moved reference to Figure 5 to the new Traditional Unicast Operation section. |
| | 13 | Added descriptions of unicast and multicast transmission modes and Figure 4, which illustrates the differences. |
| | 14 | Moved text into a new section titled "Traditional Unicast Operation". |
| | 14 | Added "Unicast Operation" to Figure 5 caption. |
| | 15 → 17 | Moved data parsing paragraph to a new section. |
| | 15 – 16 | Added a "Multicast Operation" section. |
| | 17 | Added C++ as a programming language in the programming Language section. |
| | 19 | Removed "CDS CD-1.1 software uses a one-writer many-reader paradigm for accessing a Frame Store. This design allows reliable communication of data between processes." from ¶ 2. |
| | 20 | Changed "Design Model" to "Traditional Unicast Design Model". |
| | 20 | Added "using unicast transmission technology" in ¶ 1 of the "Traditional Unicast Design Model" section. |
| | 21 – 28 | Added sections to the Design Decisions section for multicasting: " Multicast Design Model ", " Custom Multicast Solution ", " Data Provider Provides Reliable Multicasting ", " Unicast Catchup ", " Separate Multicast and Unicast Catchup Subsystems ", " Reliability Hosts for Unicast Catchup Subsystem ", " Multicast Connection Initiation ", " Unicast Catchup Connection Initiation ", and " Multicast Startup Time ". |
| | 29 | Changed the functional description of the <i>Connection Manager</i> from "supports data connections at a data consumer" to "manages data connection requests". |
| | 29 | Changed the functional description of the <i>Connection Originator</i> from "supports data connections by a data provider" to "requests data connections". |
| | 29 | Added <i>Multicast Sender</i> and <i>Multicast Receiver</i> functional descriptions. |
| | 30 | Placed text in a new "Traditional Unicast Functional Description" section. |

| Date | Page | Change |
|--------------|-----------------------------|---|
| January 2003 | 31 | Added Unicast to the processes and caption of Figure 12 for clarity. |
| | 32–34 | Added a Multicast Functional Description section. |
| | 34 | Moved descriptions of the software components from the Functional Description section to a new Software Components section. |
| | 34–35 | Changed the Connection Manager description to include multicast. |
| | 36 | Changed “agreement to exchange protocol frames” to “agreement to maintain a connection”. |
| | 36–37 | Changed the Connection Originator description: “agreement to exchange protocol frames” → “agreement to maintain a connection” description of difference between unicast and multicast added “data consumer” → “protocol peer” “Exchange Controller” → “child process” “Multicast Receiver” included as a child process. “Option Request Frames and Option Response Frames” → “connection information” |
| | 37 | Changed “For example, newly received Data Frames” to “For example, in a data forwarding configuration, newly received Data Frames”. |
| | 38 | Added a Multicast Sender description. |
| | 39 | Added a Multicast Receiver description. |
| | 43 | Added “(or UDP for multicast)” to the Interface with External Users section. |
| | 43 | Deleted the last bullet in the Interface with Operators section. |
| | 45, 135–136 | Added “Multicast Protocol” section. |
| | 46 | Added “The CDS CD-1.1 software has two operational modes, traditional unicast and multicast.” to the end of the paragraph. Added “Traditional Unicast Operation” section header. Added “ for traditional unicast operation at the IDC” to the end of the first sentence. |
| | 46 | Changed “daemon is configured to respond to the port” to “daemon responds to the port”. |

| Date | Page | Change |
|--------------|-----------------------------|--|
| January 2003 | 47 | Changed "forks an <i>Exchange Controller</i> " to "exec's an <i>Exchange Controller</i> ". |
| | 47 | Added "Traditional Unicast Operation" to the Figure 14 caption. |
| | 48 | Changed "forks" to "execs" in two places. Deleted "In the data consumer (inbound) mode <i>Exchange Controller</i> does not respond to new frame notifications. " |
| | 49 | Changed "CSS 3.0 data format using information obtained from the DBMS describing the data provider" to "CSS 3.0 data format". |
| | 49 | Changed "forks" to "exec's" in two places. |
| | 49-53 | Added "Multicast Operation" section. |
| | 53, 98-103 | Added "Multicast Sender" section. |
| | 53, 103-109 | Added "Multicast Receiver" section. |
| | 53, 109-113 | Added "Missing Frame Detector" section. |
| | 54 | Changed "data provider" to "protocol peer". |
| | 54 | Added "Authentication Certificates" (D4) to Figure 16 . |
| | 55 | Changed "Connection Request" and "Recognized protocol peer" bullets: <p>"This frame is sent via socket communications from a data provider" → "This frame is received via socket communications from a protocol peer."</p> <p>"Recognized provider data. This information is stored in database tables and is accessed with <i>libgdi</i> library functions when validating connection requests." → "Recognized protocol peer. This information may be stored in database tables and, if so, is accessed with <i>libgdi</i> library functions when validating connection requests. In the absence of a DBMS, data are maintained in files."</p> |
| | 56 | Added "Authentication Certificates" (D4) to Figure 17 . |

| Date | Page | Change |
|--------------|-----------------------|--|
| January 2003 | 57 | Revised the description of Connection Manager Processing: “Response” → “Request” “To establish a server connection <i>Connection Manager</i> obtains a list of <i>Connection Manager Server</i> hosts by using <i>Database Processing</i> ” → “For traditional unicast and unicast catchup connections, a connection server must be selected. <i>Connection Manager</i> obtains a list of <i>Connection Manager Server</i> hosts by using either <i>Database Processing</i> or configuration parameters,” Added a sentence describing how <i>Frame Processing</i> builds a Connection Response Frame and a paragraph on multicast connections |
| | 58 | Revised the description of Frame Processing: “Connection Response Frames” → “Connection Request Frames” Added “For traditional unicast and unicast catchup connections,” before the second set of bullets option size” → “option type” |
| | 59 | Deleted reference to <i>Connection Manager</i> and <i>Connection Manager Server</i> in Socket Processing. |
| | 59 | Replaced “data provider” with “protocol peer” in the second bullet of the Output section. |
| | 60 | Replaced “Connection Response” with “Connection Request” in the first paragraph of the Control section. |
| | 60–61 | Replaced “data provider” with “protocol peer” in the bullets of the Error States section. |
| | 61 | Changed “to negotiate a network connection for a requesting data provider” to “to establish traditional unicast or unicast catchup connections for a requesting protocol peer. <i>Connection Manager Server</i> is not used for multicast connections”. |
| | 62 | Added “missing sequence numbers” data store (D3) in Figure 18 . |
| | 63 | Changed the text in the Option Request Frames bullet to include a catchup operation. |
| | 63 | Replaced “data provider” with “protocol peer” in the third bullet and added a bullet on unicast catchup. |

| Date | Page | Change |
|--------------|--------------------|--|
| January 2003 | 63 | Replaced "data provider" with "protocol peer" and "providers" with "peers". |
| | 64 | Added Figure 19 . |
| | 65 | Changed <i>Connection Manager Server</i> process description to include catchup operation. |
| | 65 | Added "missing frame sequence numbers" to outputs. |
| | 65 | Changed <i>Connection Manager Server</i> output descriptions to include catchup operation. |
| | 66 | Clarified that <i>inetd</i> starts <i>Connection Manager Server</i> . |
| | 66 | Replaced "data provider" with "protocol peer" and "providers" with "peers" in the Interfaces and Error States sections. |
| | 67 | Added "unsupported option" as a reason for denying a request. |
| | 67 | Added <i>Missing Frame Detector</i> as a managed process. |
| | 67 | Redesigned Figure 20 . |
| | 69 | Changed "(stderr) output" to "(stderr) and file output". |
| | 70 | Changed "forks" to "execs". |
| | 75 | Updated the description of the <i>Connection Originator</i> to include multicast. Added "missing sequence numbers" data store and changed <i>Exchange Controller</i> to <i>child processes</i> in Figure 23 . |
| | 76 | Changed "protocol frames" to "frames" and " <i>Exchange Controller</i> " to "child processes". |
| | 77 | Added a step to the <i>Connection Originator</i> processing description. |
| | 77 | Changed " <i>Exchange Controller</i> " to "child process". |
| | 78 | Added "and <i>Multicast Receiver</i> " in two places. |
| | 78 | Added a failure possibility and changed " <i>Exchange Controller</i> " to "child process" in Error States. |
| | 79 | Added "missing sequence numbers" data store and split <i>Connection Manager Server</i> and <i>Connection Originator</i> in Figure 25 . |
| | 80 | Added "missing sequence numbers" data store and split <i>Connection Manager Server</i> and <i>Connection Originator</i> in Figure 26 . |

| Date | Page | Change |
|--------------|---------------------|---|
| January 2003 | 81 | Changed "methods of other objects to pass along information received. The methods provided" to "specific functions to process this class of frame message. The functions provided". |
| | 81 | Changed "methods" to "functions". Added "(in traditional unicast operation)". Deleted " <i>Frame Handler</i> does not resolve inbound frames (into <i>Exchange Controller</i>). Inbound frames to the host computer are resolved by <i>Frame Exchange</i> ." |
| | 81 | Added text describing unicast catchup. |
| | 81 | Changed "forks" to "execs". |
| | 82 | Added paragraph describing how <i>Exchange Controller</i> exits in unicast catchup. |
| | 83 | Changed "As part of initialization processing" to "For traditional unicast initialization". |
| | 83 | Added paragraph describing how <i>Frame handler</i> reads frames to be sent. |
| | 84 | Changed "forked" to "exec'd". |
| | 85 | Added ", for traditional unicast operation,". |
| | 85 | Changed "for" to "wants to terminate". |
| | 85 | Added " for traditional unicast operation. For unicast catchup operation, <i>Exchange Controller</i> exits after the missing frames have been sent and acknowledged." to the end of the section. |
| | 85 | Added "and the missing frame sequence numbers file" to the end of the first paragraph in the Interfaces section. |
| | 123 | Redrew Figure 37 . |
| | 136 | Moved the second paragraph of the Database Description text into a new "Database Interface" section. |
| | 137 | Moved reference to Figure 40 from the end of the second paragraph to the first sentence of this section. |
| | 142 | Updated the introductory paragraph for the Connection Requirements section. |
| | 144 | Added descriptions of traditional unicast and unicast catchup operation to the Connection Originator Requirements. |

| Date | Page | Change |
|--------------|-------------------------|--|
| January 2003 | 145 | Added "in traditional unicast and unicast catchup operation". |
| | 146 | Changed "by a socket (TCP or UDP) " to "by a TCP/IP socket". |
| | 154 | Added system delivery and performance requirements 95a-95g . |
| | 155 | Added system requirements for multicast and unicast catchup capabilities (102 , M-1 through M-13 , and C-1 through C-2). |
| | 157 | Clarified how the requirement was fulfilled in requirements 4 and 5 . |
| | 159 | Added the caveat "In traditional unicast and unicast catchup operation," to the How Fulfilled column for requirement 20 . |
| | 160 | Added the caveat "In unicast operation," to the How Fulfilled column for requirements 22 and 24 . |
| | 172-173 | Added system delivery and performance requirements 95a-95g . |
| | 175-177 | Added system requirements for multicast and unicast catchup capabilities (102 , M-1 through M-13 , and C-1 through C-2). |
| | 179 | Added reference for [Aga01]. Updated references for [DOD94a] and [DOD94b]. Changed reference to a later document version for [IDC3.4.1Rev3], [IDC3.4.2Rev0.1], [IDC3.4.3Rev0.2], and [IDC5.1.1Rev3] |
| | 180 | Added reference for [SAIC-01/3068] |
| | G1 | Added definition for AckNack. Simplified the definition of child process. |
| | G2 | Changed the punctuation for the definition of COTS. Added definition for cron. |
| | G3 | Changed the punctuation for the definition of CTBTO and CSP. |
| | G4 | Expanded the definition of fork and changed the punctuation for the definition of FTP. |
| | G5 | Added definitions for IGMP and inetd. Capitalized Protocol in the definition of IP. |

| Date | Page | Change |
|--------------|--------------------|--|
| January 2003 | G6 | Added definitions for MTU, multicast, and network. Shortened definition of parent process. |
| | G7 | Added definitions for PNack, protocol, and spawn. |
| | G9 | Added definitions for SSL, unicast, and WAN. Changed punctuation in the definition of Web. |

Continuous Data Subsystem CD-1.1

CONTENTS

| | |
|---|----|
| <u>About this Document</u> | i |
| ■ <u>PURPOSE</u> | ii |
| ■ <u>SCOPE</u> | iv |
| ■ <u>AUDIENCE</u> | iv |
| ■ <u>RELATED INFORMATION</u> | iv |
| ■ <u>USING THIS DOCUMENT</u> | v |
| <u>Conventions</u> | vi |
| <u>Chapter 1: Overview</u> | 1 |
| ■ <u>INTRODUCTION</u> | 2 |
| ■ <u>FUNCTIONALITY</u> | 6 |
| ■ <u>IDENTIFICATION</u> | 6 |
| ■ <u>BACKGROUND AND HISTORY</u> | 7 |
| ■ <u>OPERATING ENVIRONMENT</u> | 7 |
| <u>Hardware</u> | 7 |
| <u>Network Environment</u> | 8 |
| <u>Commercial-Off-The-Shelf Software</u> | 9 |
| <u>Chapter 2: Architectural Design</u> | 11 |
| ■ <u>CONCEPTUAL DESIGN</u> | 12 |
| <u>Traditional Unicast Operation</u> | 14 |
| <u>Multicast Operation</u> | 15 |
| <u>Data Parsing</u> | 17 |
| ■ <u>DESIGN DECISIONS</u> | 17 |
| <u>Programming Language</u> | 17 |
| <u>Global Libraries</u> | 17 |
| <u>Database</u> | 18 |

| | |
|---|----|
| Interprocess Communication (IPC) | 18 |
| File System | 19 |
| Traditional Unicast Design Model | 20 |
| Multicast Design Model | 21 |
| Custom Multicast Solution | 21 |
| Data Provider Provides Reliable Multicasting | 21 |
| Unicast Catchup | 22 |
| Separate Multicast and Unicast Catchup Subsystems | 23 |
| Reliability Hosts for Unicast Catchup Subsystem | 24 |
| Multicast Connection Initiation | 26 |
| Unicast Catchup Connection Initiation | 26 |
| Multicast Startup Time | 27 |
| Database Schema Overview | 28 |
| ■ FUNCTIONAL DESCRIPTION | 29 |
| Traditional Unicast Functional Description | 30 |
| Multicast Functional Description | 32 |
| ■ SOFTWARE COMPONENTS | 34 |
| Connection Manager | 34 |
| Data Center Manager | 35 |
| Connection Originator | 36 |
| Exchange Controller | 37 |
| Frame Exchange | 38 |
| Multicast Sender | 38 |
| Multicast Receiver | 39 |
| Data Parser | 40 |
| Frame Store Stager | 41 |
| Authentication | 41 |
| ■ INTERFACE DESIGN | 42 |
| Interface with Other IDC Systems | 42 |
| Interface with External Users | 43 |
| Interface with Operators | 43 |
| Chapter 3: Detailed Design | 45 |
| ■ DATA FLOW MODEL | 46 |

| | |
|---|----|
| Traditional Unicast Operation | 46 |
| Multicast Operation | 49 |
| Multicast Subsystem | 51 |
| Unicast Catchup Subsystem | 51 |
| ■ PROCESSING UNITS | 53 |
| Connection Manager | 54 |
| Input/Processing/Output | 54 |
| Output | 59 |
| Control | 60 |
| Interfaces | 60 |
| Error States | 60 |
| Connection Manager Server | 61 |
| Input/Processing/Output | 62 |
| Control | 66 |
| Interfaces | 66 |
| Error States | 66 |
| Data Center Manager | 67 |
| Input/Processing/Output | 69 |
| Interfaces | 72 |
| Error States | 74 |
| Connection Originator | 75 |
| Input/Processing/Output | 77 |
| Control | 77 |
| Interfaces | 77 |
| Error States | 78 |
| Exchange Controller | 79 |
| Controller Executive | 81 |
| Exchange Interface | 82 |
| Frame Handler | 83 |
| Input/Processing/Output | 84 |
| Control | 84 |
| Interfaces | 85 |
| Error States | 87 |

| | |
|---|-----|
| Frame Exchange | 88 |
| Main Loop | 90 |
| Time Counter | 90 |
| Heartbeat | 91 |
| Message Sender | 91 |
| Sender | 92 |
| Frame I/O | 92 |
| Processing Lists | 92 |
| Input/Processing/Output | 94 |
| Control | 95 |
| Interfaces | 96 |
| Error States | 97 |
| Multicast Sender | 98 |
| Processing | 99 |
| Error States | 103 |
| Multicast Receiver | 103 |
| Processing | 104 |
| Error States | 108 |
| Missing Frame Detector | 109 |
| Processing | 109 |
| Input/Processing/Output | 111 |
| Control | 112 |
| Interfaces | 112 |
| Error States | 112 |
| Data Parser | 113 |
| DLParse Exec | 115 |
| Process Loop | 117 |
| Process Frame | 119 |
| Input | 119 |
| Output | 120 |
| Control | 120 |
| Interfaces | 121 |
| Error States | 121 |
| Frame Store Stager | 122 |

| | | |
|--|--|-----|
| | Input/Processing/Output | 123 |
| | Control | 124 |
| | Interfaces | 125 |
| | Error States | 125 |
| | Protocol Checker | 126 |
| | Input/Processing/Output | 127 |
| | Control | 127 |
| | Interfaces | 127 |
| | Error States | 128 |
| | libfs | 128 |
| | Input/Processing/Output | 129 |
| | Control | 131 |
| | Interfaces | 131 |
| | Error States | 133 |
| | libcdo | 133 |
| | Input/Processing/Output | 134 |
| | Control | 134 |
| | Error States | 135 |
| | ■ MULTICAST PROTOCOL | 135 |
| | Data Packet Format | 135 |
| | PNack Packet Format | 136 |
| | ■ DATABASE DESCRIPTION | 136 |
| | Database Interface | 136 |
| | Database Design | 137 |
| | Chapter 4: Requirements | 141 |
| | ■ INTRODUCTION | 142 |
| | ■ FUNCTIONAL REQUIREMENTS | 142 |
| | Connection Manager Requirements | 142 |
| | Data Center Manager Requirements | 143 |
| | Connection Originator Requirements | 144 |
| | Exchange Controller Requirements | 145 |
| | Frame Exchange Requirements | 146 |
| | Data Parser Requirements | 147 |

| | |
|--|-----|
| Frame Store Stager Requirements | 148 |
| Authentication Signing Requirements | 149 |
| Signature Authentication Requirements | 150 |
| Frame Store Requirements | 151 |
| libcdo Requirements | 152 |
| ■ SYSTEM REQUIREMENTS | 153 |
| Multicast Subsystem Requirements | 155 |
| Unicast Catchup Subsystem Requirements | 156 |
| ■ REQUIREMENTS TRACEABILITY | 156 |
| References | 179 |
| Glossary | G1 |
| Index | I1 |

Continuous Data Subsystem CD-1.1

FIGURES

| | | |
|------------|--|----|
| FIGURE 1. | IDC SOFTWARE CONFIGURATION HIERARCHY | 3 |
| FIGURE 2. | RELATIONSHIP OF CONTINUOUS DATA SUBSYSTEM CD-1.1 TO OTHER SOFTWARE UNITS OF DATA SERVICES CSCI | 5 |
| FIGURE 3. | REPRESENTATIVE HARDWARE CONFIGURATION FOR CDS CD-1.1 | 8 |
| FIGURE 4. | UNICAST VERSUS MULTICAST COMMUNICATION | 13 |
| FIGURE 5. | OBJECT MODEL OF CDS CD-1.1 UNICAST OPERATION | 14 |
| FIGURE 6. | OBJECT MODEL OF CDS CD-1.1 MULTICAST OPERATION | 15 |
| FIGURE 7. | RELIABLE MULTICAST SUBSYSTEM | 22 |
| FIGURE 8. | SEPARATE MULTICAST AND UNICAST CATCHUP SUBSYSTEMS | 23 |
| FIGURE 9. | RELIABILITY HOSTS FOR UNICAST CATCHUP SUBSYSTEM | 25 |
| FIGURE 10. | MULTICAST CONNECTION INITIATION | 26 |
| FIGURE 11. | UNICAST CATCHUP CONNECTION INITIATION | 27 |
| FIGURE 12. | FUNCTIONAL DESIGN OF CDS CD-1.1 TRADITIONAL UNICAST OPERATION | 31 |
| FIGURE 13. | FUNCTIONAL DESIGN OF CDS CD-1.1 MULTICAST OPERATION | 33 |
| FIGURE 14. | DATA FLOW MODEL OF CDS CD-1.1 TRADITIONAL UNICAST OPERATION | 47 |
| FIGURE 15. | DATA FLOW MODEL OF CDS CD-1.1 MULTICAST OPERATION | 50 |
| FIGURE 16. | CONNECTION MANAGER CONTEXT | 54 |
| FIGURE 17. | CONNECTION MANAGER COMPONENTS | 56 |
| FIGURE 18. | CONNECTION MANAGER SERVER CONTEXT | 62 |
| FIGURE 19. | CONNECTION MANAGER SERVER COMPONENTS | 64 |
| FIGURE 20. | DATA CENTER MANAGER CONTEXT | 67 |
| FIGURE 21. | DATA CENTER MANAGER PROCESSING COMPONENTS | 68 |
| FIGURE 22. | DATA CENTER MANAGER INTERNAL CONTROL FLOW | 74 |
| FIGURE 23. | CONNECTION ORIGINATOR CONTEXT | 75 |
| FIGURE 24. | CONNECTION ORIGINATOR INTERNAL DATA AND CONTROL FLOW | 76 |

| | | |
|-----------------------------------|---|-----|
| <u>FIGURE 25.</u> | <u>EXCHANGE CONTROLLER CONTEXT</u> | 79 |
| <u>FIGURE 26.</u> | <u>EXCHANGE CONTROLLER DATA FLOW</u> | 80 |
| <u>FIGURE 27.</u> | <u>FRAME EXCHANGE CONTEXT</u> | 88 |
| <u>FIGURE 28.</u> | <u>FRAME EXCHANGE COMPONENTS</u> | 89 |
| <u>FIGURE 29.</u> | <u>MULTICAST SENDER CONTEXT</u> | 98 |
| <u>FIGURE 30.</u> | <u>MULTICAST SENDER DATA AND CONTROL FLOW</u> | 99 |
| <u>FIGURE 31.</u> | <u>MULTICAST RECEIVER CONTEXT</u> | 104 |
| <u>FIGURE 32.</u> | <u>MULTICAST RECEIVER DATA AND CONTROL FLOW</u> | 105 |
| <u>FIGURE 33.</u> | <u>MISSING FRAME DETECTOR CONTEXT</u> | 109 |
| <u>FIGURE 34.</u> | <u>MISSING FRAME DETECTOR DATA AND CONTROL FLOW</u> | 110 |
| <u>FIGURE 35.</u> | <u>DATA PARSER CONTEXT</u> | 114 |
| <u>FIGURE 36.</u> | <u>DATA PARSER DATA FLOW</u> | 115 |
| <u>FIGURE 37.</u> | <u>FRAME STORE STAGER CONTEXT</u> | 123 |
| <u>FIGURE 38.</u> | <u>FRAME STORE STAGER DATA FLOW</u> | 124 |
| <u>FIGURE 39.</u> | <u>PROTOCOL CHECKER CONTEXT</u> | 126 |
| <u>FIGURE 40.</u> | <u>CDS CD-1.1 TABLE RELATIONSHIPS</u> | 138 |

Continuous Data Subsystem CD-1.1

TABLES

| | | |
|-----------------------------------|--|-----|
| <u>TABLE I:</u> | <u>DATA FLOW SYMBOLS</u> | vi |
| <u>TABLE II:</u> | <u>ENTITY-RELATIONSHIP SYMBOLS</u> | vii |
| <u>TABLE III:</u> | <u>TYPOGRAPHICAL CONVENTIONS</u> | vii |
| <u>TABLE 1:</u> | <u>DATABASE TABLES USED BY CDS CD-1.1</u> | 28 |
| <u>TABLE 2:</u> | <u>DATA CENTER MANAGER JOB TEMPLATE ATTRIBUTES</u> | 70 |
| <u>TABLE 3:</u> | <u>DATA CENTER MANAGER EVENTS</u> | 71 |
| <u>TABLE 4:</u> | <u>FORMAT OF DATA PACKET</u> | 135 |
| <u>TABLE 5:</u> | <u>FORMAT OF PNACK PACKET</u> | 136 |
| <u>TABLE 6:</u> | <u>DETAILED DATABASE USAGE BY CDS CD-1.1</u> | 139 |
| <u>TABLE 7:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: CONNECTION MANAGER</u> | 157 |
| <u>TABLE 8:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: DATA CENTER MANAGER</u> | 158 |
| <u>TABLE 9:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: CONNECTION ORIGINATOR</u> | 159 |
| <u>TABLE 10:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: EXCHANGE CONTROLLER</u> | 160 |
| <u>TABLE 11:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: FRAME EXCHANGE</u> | 161 |
| <u>TABLE 12:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: DATA PARSER</u> | 163 |
| <u>TABLE 13:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: FRAME STORE STAGER</u> | 165 |
| <u>TABLE 14:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: AUTHENTICATION SIGNING</u> | 166 |
| <u>TABLE 15:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS: SIGNATURE AUTHENTICATION</u> | 167 |

| | | |
|------------------|---|-----|
| <u>TABLE 16:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS:</u> <u>FRAME STORE</u> | 168 |
| <u>TABLE 17:</u> | <u>TRACEABILITY OF FUNCTIONAL REQUIREMENTS:</u> <u>LIBCDO REQUIREMENTS</u> | 170 |
| <u>TABLE 18:</u> | <u>TRACEABILITY OF SYSTEM REQUIREMENTS</u> | 171 |

About this Document

This chapter describes the organization and content of the document and includes the following topics:

- [Purpose](#)
- [Scope](#)
- [Audience](#)
- [Related Information](#)
- [Using this Document](#)

About this Document

PURPOSE

This document describes the design and requirements of the Continuous Data Subsystem CD-1.1 (CDS CD-1.1) software of the International Data Centre (IDC). The software is a computer software component (CSC) of the Data Services Computer Software Configuration Item (CSCI). This document provides a basis for implementing, supporting, and testing the software.

This document is Revision 1 of *Continuous Data Subsystem CD-1.1*, originally published in March 2001. The following changes were made to incorporate multicast capability:

- General
The publication date and document number in the footers was changed to reflect the revision. Change bars (vertical lines) were added to identify new or revised material. The Notice Page was changed to include a new SAIC number. Terminology changes were made throughout to reflect traditional unicast capability (original functionality) and new multicast and unicast catchup capabilities.
- [About this Document](#)
A description of the changes to the document was added.
- [Chapter 1: Overview](#)
Removed the Status of Development section. Added three components for multicasting to [Identification](#). Added [Network Environment](#) to [Operating Environment](#). Minor revisions were made throughout to incorporate additional capability.

■ [Chapter 2: Architectural Design](#)

Unicast and multicast transmission modes and operation were added to [Conceptual Design](#). Decisions relating to multicasting were added to [Design Decisions](#). [Multicast Functional Description](#) was added to [Functional Description](#). [Software Components](#) was elevated to its own major section and *Multicast Sender* and *Multicast Receiver* were added. *Connection Originator*, *Connection Manager*, *Exchange Controller*, and *Frame Exchange* were modified to reflect new functionality related to multicast operation.

■ [Chapter 3: Detailed Design](#)

[Data Flow Model](#) was revised to include [Multicast Operation](#) consisting of multicast and unicast catchup subsystems. Three components were added to [Processing Units](#), *Multicast Sender*, *Multicast Receiver*, and *Missing Frame Detector*. *Connection Originator*, *Connection Manager*, *Connection Manager Server*, *Exchange Controller*, and *Frame Exchange* were modified to reflect new functionality related to multicast operation. [Multicast Protocol](#) was added containing descriptions of UDP data and PNack packets.

■ [Chapter 4: Requirements](#)

System requirements were added for multicasting and unicast catchup capabilities ([102](#), [M-1](#) through [M-13](#), and [C-1](#) through [C-2](#)). System delivery and performance requirements were added ([95a](#) through [95g](#)). Component descriptions and “How Fulfilled” information in the requirements traceability matrices were modified to accommodate new system requirements.

■ [References](#)

References were added for [\[Aga01\]](#) and [\[SAIC-01/3068\]](#). Other reference were updated to reflect the current versions.

■ [Glossary](#)

Terms and definitions were added for *cron*, IGMP, *inetd*, MTU, multicast, network, PNack, protocol, spawn, SSL, UDP, unicast, and WAN.

SCOPE

Continuous Data Subsystem CD-1.1 software is identified as follows:

Title: Continuous Data Subsystem CD-1.1

Abbreviation: *CDS CD-1.1*

This document describes the architectural and detailed design of the *CDS CD-1.1* software including its functionality, components, data structures, high-level interfaces, method of execution, and underlying hardware. Additionally, this document specifies the requirements of the software and its components. This information is modeled on the Data Item Description for Software Design Descriptions [\[DOD94a\]](#) and Software Requirements Specification [\[DOD94b\]](#). Software that supports the CD-1.0 (Alpha) protocol is not addressed by this document.

AUDIENCE

This document is intended for all engineering and management staff concerned with the design and requirements of all IDC software in general and of *CDS CD-1.1* in particular. The detailed descriptions are intended for programmers who develop, test, or maintain *CDS CD-1.1*.

RELATED INFORMATION

The following documents complement this document:

- *Formats and Protocols for Continuous Data CD-1.1* [\[IDC3.4.3Rev0.2\]](#)
- *Continuous Data Subsystem CD-1.1 Software User Manual* [\[IDC6.5.18\]](#)

See [References](#) for a list of documents that supplement this document.

USING THIS DOCUMENT

This document is part of the overall documentation architecture for the IDC. It is part of the Software category, which describes the design of the software. This document is organized as follows:

- [Chapter 1: Overview](#)

This chapter provides a high-level view of *CDS CD-1.1*, including its functionality, components, background, status of development, and current operating environment.

- [Chapter 2: Architectural Design](#)

This chapter describes the architectural design of *CDS CD-1.1*, including its conceptual design, design decisions, functions, and interface design.

- [Chapter 3: Detailed Design](#)

This chapter describes the detailed design of *CDS CD-1.1*, including its data flow, software units, and database design.

- [Chapter 4: Requirements](#)

This chapter describes the general, functional, and system requirements for *CDS CD-1.1*. Traceability tables define how the general and functional requirements are met.

- [References](#)

This section lists the sources cited in this document.

- [Glossary](#)

This section defines the terms, abbreviations, and acronyms used in this document.

- [Index](#)

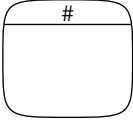


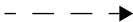

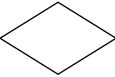
This section lists topics and features provided in the document along with page numbers for reference.

▼ About this Document

Conventions

This document uses a variety of conventions, which are described in the following tables. [Table I](#) shows the conventions for data flow diagrams. [Table II](#) shows the conventions for entity-relationship diagrams. [Table III](#) lists typographical conventions.

TABLE I: DATA FLOW SYMBOLS

| Description ¹ | Symbol |
|---|---|
| process |  |
| external source or sink of data |  |
| data store D = disk store Db = database store |  |
| control flow |  |
| data flow |  |
| decision |  |

1. Symbols in this table are based on Gane-Sarson conventions.

TABLE II: ENTITY-RELATIONSHIP SYMBOLS











| Description | Symbol | | | | | | | |
|--|---|-----------|--|--|--------------------|--------------------|-----|--------------------|
| One A maps to one B. | A  B | | | | | | | |
| One A maps to zero or one B. | A  B | | | | | | | |
| One A maps to many Bs. | A  B | | | | | | | |
| One A maps to zero or many Bs. | A  B | | | | | | | |
| database table | <table><tr><th>tablename</th></tr><tr><td> <i>primary key</i></td></tr><tr><td> <i>foreign key</i></td></tr><tr><td><i>attribute 1</i></td></tr><tr><td><i>attribute 2</i></td></tr><tr><td>...</td></tr><tr><td><i>attribute n</i></td></tr></table> | tablename |  <i>primary key</i> |  <i>foreign key</i> | <i>attribute 1</i> | <i>attribute 2</i> | ... | <i>attribute n</i> |
| tablename | | | | | | | | |
|  <i>primary key</i> | | | | | | | | |
|  <i>foreign key</i> | | | | | | | | |
| <i>attribute 1</i> | | | | | | | | |
| <i>attribute 2</i> | | | | | | | | |
| ... | | | | | | | | |
| <i>attribute n</i> | | | | | | | | |

TABLE III: TYPOGRAPHICAL CONVENTIONS

| Element | Font | Example |
|---|----------------|---|
| database tables | bold | dlfile |
| database attributes | <i>italics</i> | <i>dlid</i> |
| processes, software units, and libraries | | <i>ConnMgr</i> |
| user-defined arguments and variables used in parameter (par) files or program command lines | | run_idc_dcmgr <hostname> |
| titles of documents | | <i>Continuous Data Subsystem CD-1.1</i> |
| computer code and output | courier | [info]:Parameter inetd - 1 |
| filenames, directories, and websites | | DLlog.log |
| text that should be typed exactly as shown | | ps -fu <cds-user-name> |

Chapter 1: Overview

This chapter provides a general overview of the *CDS CD-1.1* software and includes the following topics:

- [Introduction](#)
- [Functionality](#)
- [Identification](#)
- [Background and History](#)
- [Operating Environment](#)

Chapter 1: Overview

INTRODUCTION

The software of the IDC acquires time-series and radionuclide data from stations of the International Monitoring System (IMS) and other locations. These data are passed through automatic and interactive processing, which culminates in the estimation of location and in the origin time of events (earthquakes, volcanic eruptions, and so on) in the earth, including its oceans and atmosphere. The results of the processing are distributed to States Parties and other users by various means. Approximately one million lines of developmental software are spread across six computer software configuration items (CSCIs) of the software architecture. One additional CSCI is devoted to run-time data of the software. [Figure 1](#) shows the logical organization of the IDC software. The Data Services CSCI receives, stores, and distributes data through the following computer software components (CSCs):

- Continuous Data Subsystem

This software acquires time-series data according to two standard protocols and forwards the data to external users ([\[IDC3.4.2Rev0.1\]](#) and [\[IDC3.4.3Rev0.2\]](#)).

- Message Subsystem

This software exchanges data in response to user requests. The data are formatted according to a standard protocol and exchanged through UNIX mail (see [\[IDC3.4.1Rev3\]](#)). This software also provides the interface to mail for the Retrieve and Subscription Subsystems.

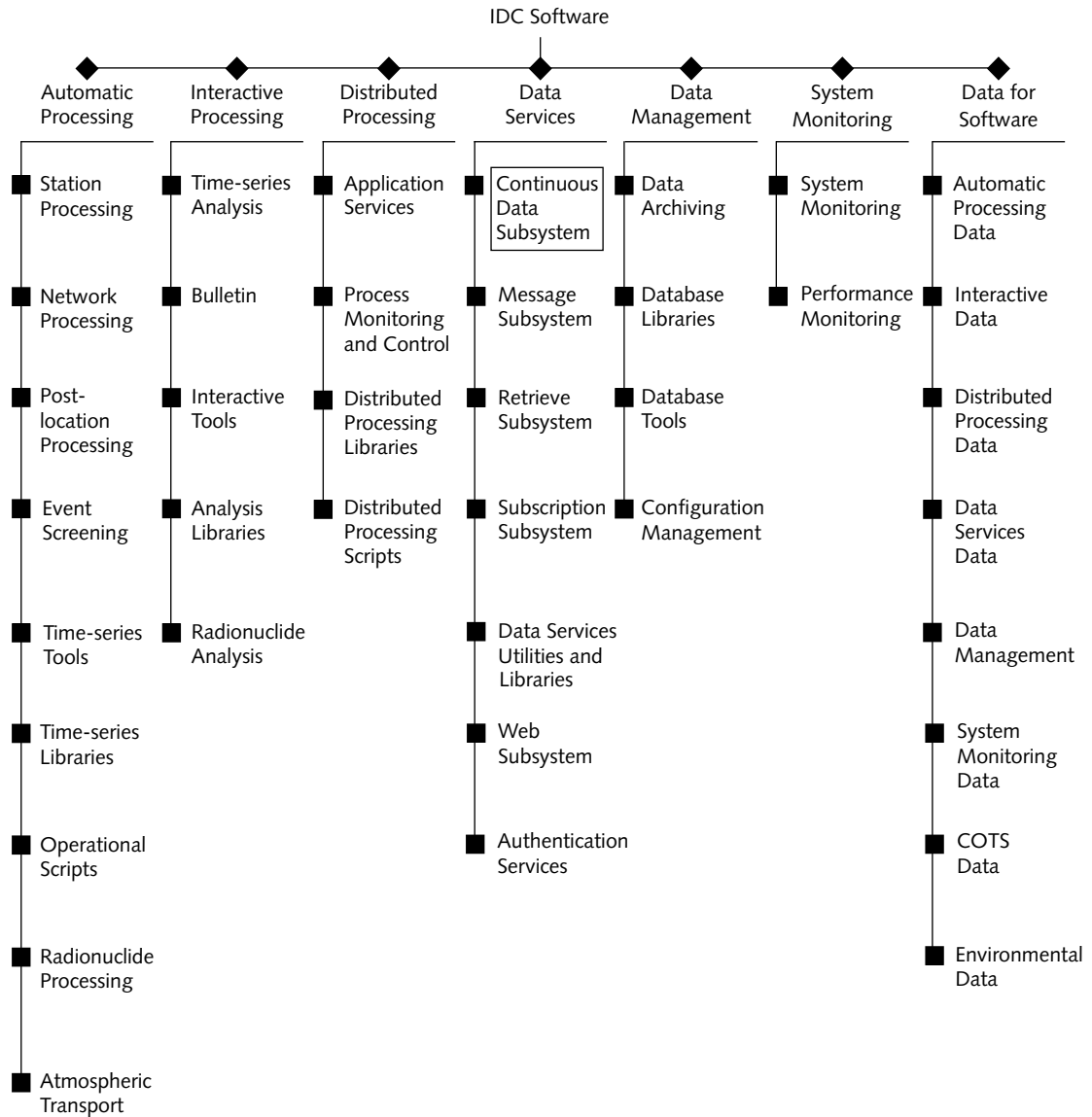


FIGURE 1. IDC SOFTWARE CONFIGURATION HIERARCHY

▼ Overview

■ Retrieve Subsystem

This software prepares messages, formatted according to the standard protocol, that retrieve segments of data from stations of the IMS auxiliary seismic network (see [\[IDC3.4.1Rev3\]](#)). The software also parses the response messages. The Message Subsystem exchanges the messages.

■ Subscription Subsystem

This software maintains a subscriber database and prepares the regular data products for delivery to subscribers. The Message Subsystem receives the subscription requests and delivers the subscription products.

■ Data Services Utilities and Libraries

This software consists of utilities used by data services operators and libraries common to data services.

■ Web Subsystem

This software runs the IDC website.

■ Authentication Services

This software provides data signing and verification services to Data Services subsystems using the Digital Signature Algorithm (DSA).

[Figure 2](#) shows the relationship of the *CDS CD-1.1* to the other components of the other CSCIs. [Figure 2](#) indicates that the *CDS CD-1.1* is responsible for receiving CD-1.1 data from IMS stations and the *CDS CD-1.0* is responsible for receiving CD-1.0 data. In this respect the Continuous Data Subsystem (CDS) is a “front-end” to the Data Services CSCI and the IMS by providing the processing for acquiring data. The figure also shows the CDS as the component that is responsible for forwarding CD-1.1 data to data centers of State Signatories. The Data Services CSCI contains other components besides CDS that are capable of receiving and sending data, most notably the Message Subsystem. However, the CDS is concerned with the near-real-time stream of IMS data, whereas other components are more concerned with asynchronous request/response type flows.

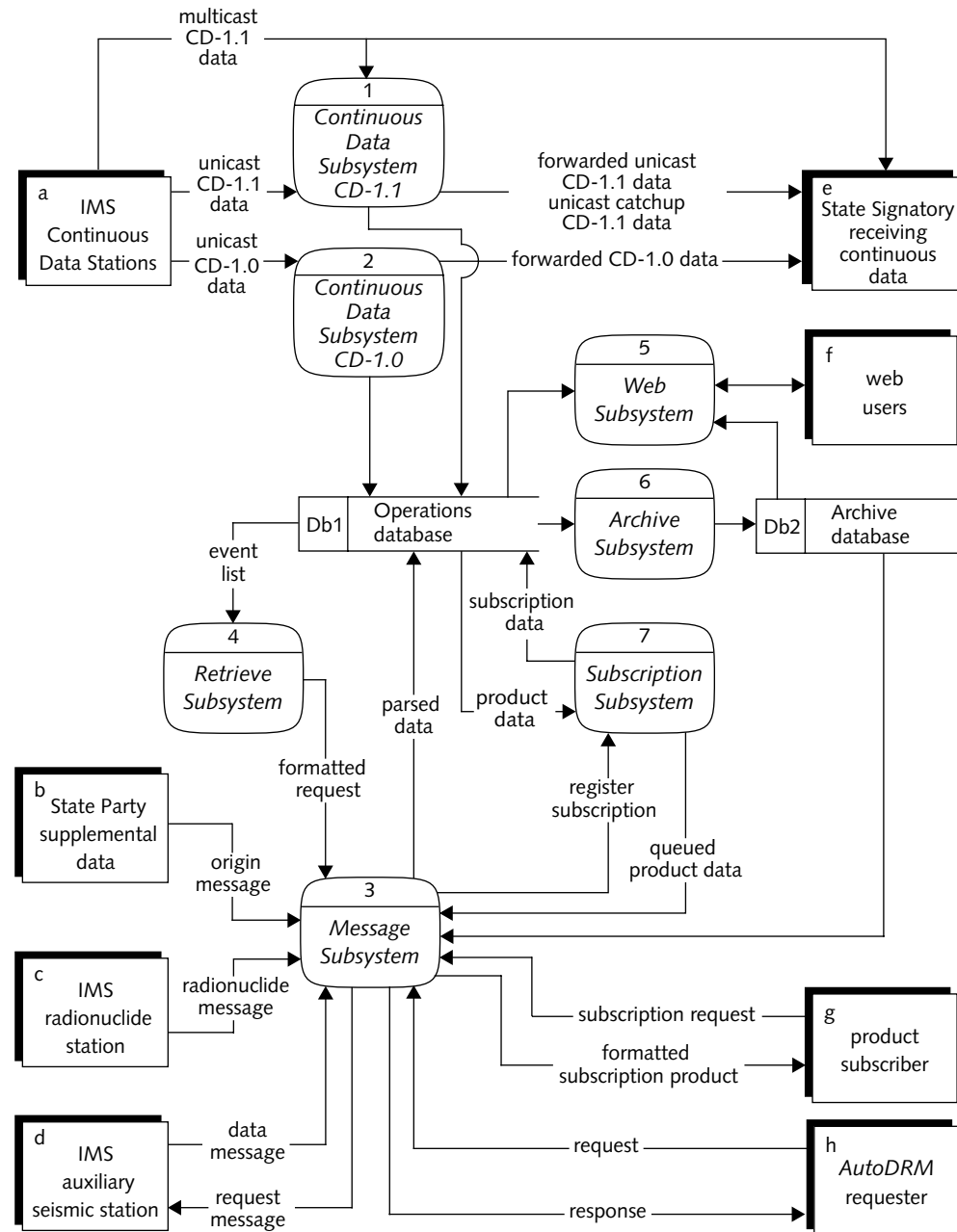


FIGURE 2. RELATIONSHIP OF CONTINUOUS DATA SUBSYSTEM CD-1.1 TO OTHER SOFTWARE UNITS OF DATA SERVICES CSCI

FUNCTIONALITY

The *CDS CD-1.1* provides data centers with the ability to receive time-series data from providing sources via unicast or multicast CD-1.1 protocol. After receipt, the data are authenticated, forwarded, parsed, and stored. Authentication checks the validity of the data using the digital signature sent in the CD-1.1 frames and public keys available for those data. Forwarding sends data packets/frames received from data providers to other data centers (data consumers). Parsing converts the CD-1.1 frames into the CSS 3.0 format for use by other processing components of the system.

IDENTIFICATION

CDS CD-1.1 components are identified as follows:

- *Connection Manager (ConnMgr)*
- *Data Center Manager (ForeMan)*
- *Connection Originator (ConnOrig)*
- *Exchange Controller (ExCltr)*
- *Frame Exchange (FrameEx)*
- *Multicast Sender (MCastProvider)*
- *Multicast Receiver (MCastConsumer)*
- *Missing Frame Detector (FrameAudit)*
- *Data Parser (DLParse)*
- *Frame Store Stager (FSstage)*
- *Protocol Checker (ProtoCheck)*
- *libfs*
- *libcdo*

BACKGROUND AND HISTORY

The software to support the CD-1.1 protocol was developed by Science Application International Corporation (SAIC) as a reference implementation of a Continuous Data Subsystem for handling the new generation protocol. The *CDS CD-1.1* design for the Data Services CSCI was developed from that reference implementation in 1999–2000. In 2002 multicast capability was added to *CDS CD-1.1*. *CDS CD-1.1* was first delivered for operational use in the summer of 2000 as an element of the PIDC at the Center for Monitoring Research (CMR) in Arlington, Virginia, U.S.A. *CDS CD-1.1* was delivered to the IDC of the Comprehensive Nuclear-Test-Ban Treaty Organization (CTBTO) IDC in Vienna, Austria, in December 2000. At the time of the initial delivery no IMS sites were capable of participating in the CD-1.1 protocol. However, it is expected that all new sites and sites undergoing upgrades will use the CD-1.1 protocol. As a result, migration to the use of the CD-1.1 protocol and the *CDS CD-1.1* will be gradual.

OPERATING ENVIRONMENT

The following paragraphs describe the hardware, network environment, and commercial-off-the-shelf (COTS) software required to operate the *CDS CD-1.1*.

Hardware

The *CDS CD-1.1* software is designed to run on UNIX-type computers such as those produced by Sun Microsystems with the Solaris 2.7 operating system. A *CDS CD-1.1* host requires a minimum of 128 MB of memory and a minimum of 9 GB of local disk storage. The software requires processing capabilities equivalent to a Sun Microsystems Ultra 60 with two 250 MHz processors. *CDS CD-1.1* hosts also require connection to a Local Area Network (LAN) on a minimum of a 10-base T line (providing a minimum of 10 M-bits per second transmission rates).

The *CDS CD-1.1* is predominately concerned with the movement of data, and there are no requirements to have graphical display hardware. However, as with most software systems, some pieces of information are desirable to monitor. For this purpose, *CDS CD-1.1* hosts must be accessible (via the LAN) to a monitor. *CDS*

▼ Overview

CD-1.1 requires the availability of a Database Management System (DBMS) such as ORACLE 8i via the LAN. [Figure 3](#) shows a representative hardware configuration with processing distributed over three hosts (CDS-A, CDS-B, and CDS-C), access to a DBMS, and a monitor.

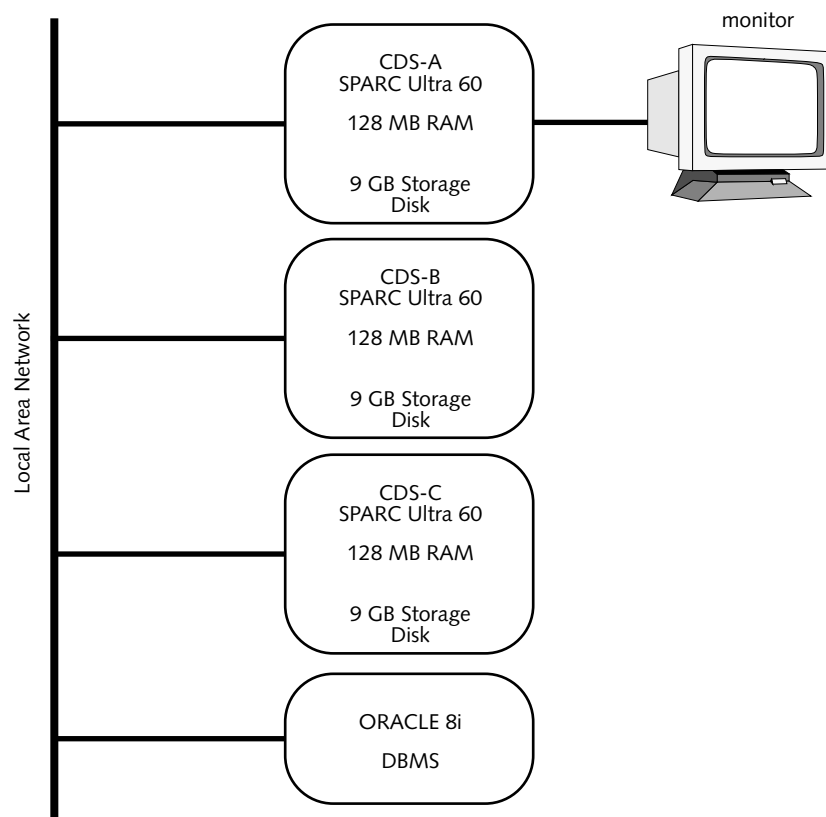


FIGURE 3. REPRESENTATIVE HARDWARE CONFIGURATION FOR CDS CD-1.1

Network Environment

For unicast operation, the implementation connects to the LAN via ethernet using the Transmission Control Protocol/Internet Protocol (TCP/IP). For multicast operation, the implementation uses the User Datagram Protocol (UDP) IP protocol. TCP/IP provides guaranteed data delivery and data order. Sender and receiver

rates are matched. TCP/IP also provides network congestion control by automatically backing off on transmission rates to accommodate other network traffic. Conversely, UDP provides little error recovery making reliable data delivery more of a challenge.

Routers on the LAN and those that interface to the WAN need to be multicast capable. The UDP and TCP/IP protocols may co-exist on the same LAN connections. However, there is a danger of UDP overwhelming the network since it lacks the adaptive transmission rate control provided by TCP. This danger is addressed by using rate-based flow control for the UDP-based multicasting traffic. In multicast operation the UDP protocol is used both for the multicasting of data packets and for the unicasting of negative acknowledgments. The negative acknowledgment and other mechanisms provide strong best-effort reliability for multicast operation. At the network layer, multicast group membership is managed by the standard IGMP (Internet Group management) protocol that operates between CDS hosts and their LAN routers.

Commercial-Off-The-Shelf Software

CDS CD-1.1 software requires the use of a DBMS and has been developed using the ORACLE 8i system. ORACLE 8i is used without modification in its standard configuration and is available from licensed vendors. The DBMS is used by the *CDS CD-1.1* for two objectives: to simplify the management of incoming connections and to manage the descriptions of received (and parsed) time-series signal data.

The *OpenSSL* library is used to digitally sign and authenticate frames by the *CDS CD-1.1*. This library is publicly available via File Transfer Protocol (FTP) download on the Internet.

Chapter 2: Architectural Design

This chapter describes the architectural design of the *CDS CD-1.1* and includes the following topics:

- [Conceptual Design](#)
- [Design Decisions](#)
- [Functional Description](#)
- [Software Components](#)
- [Interface Design](#)

Chapter 2: Architectural Design

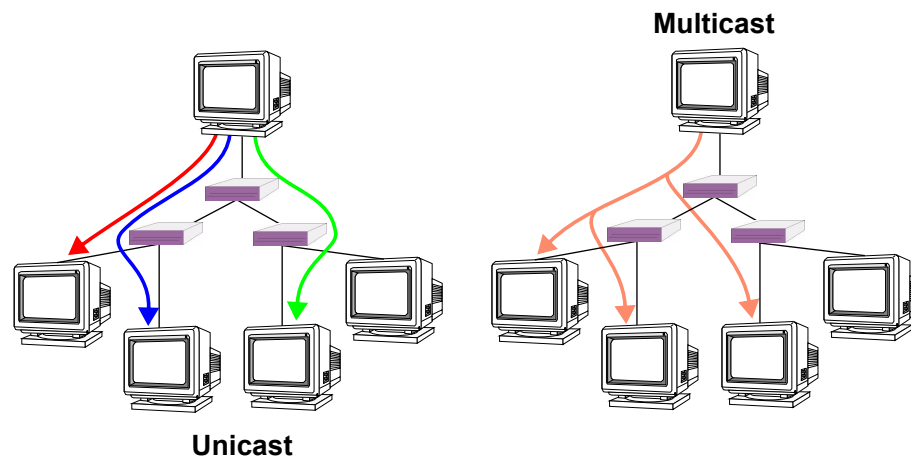
CONCEPTUAL DESIGN

The CD-1.1 protocol [\[IDC3.4.3Rev0.2\]](#) is implemented by the Continuous Data Subsystem. The CD-1.1 protocol is a set of formats, protocols, and policies that define how to send continuous time-series data from one computer to another. Physical transmission of data may occur over traditional land lines, through radio frequency (RF) connections, and via satellite links. Transmissions over these physical media are in the form of Data Frames constructed and exchanged according to the protocol defined in [\[IDC3.4.3Rev0.2\]](#). In the context of nuclear treaty monitoring, these data are acquired from seismic, hydroacoustic, infrasonic, or other environmental sensors. *CDS CD-1.1* application software executes at the IDC and at the Prototype International Data Centre (PIDC). *CDS CD-1.1* software is responsible for acquiring CD-1.1 data from providers, forwarding data to subscribing National Data Centers (NDCs), and making data available for signal processing and analysis software at the data center. The CDS also supports the CD-1.0 protocol [\[IDC3.4.2Rev0.1\]](#). The *CDS CD-1.1* software is cooperative and unobtrusive with the operation of CD-1.0 software components and vice versa. The components of the CDS that support CD-1.0 protocol are not addressed by this document.

The design of the *CDS CD-1.1* is based on the concept of providing accountable, robust handling of protocol frames both as a data consumer and as a data provider. Pursuit of this objective has produced an architecture that provides transactional processing among independent components. In particular, data receiving, parsing, and forwarding activities are decoupled from each other in this design. Decoupling allows intermittent failures in one processing element of the *CDS CD-1.1* to occur and be resolved without impacting others.

The conceptual design of the *CDS CD-1.1* is that of a hands-off system supporting connections for data providers and data consumers. The software requires that configuration information be provided to describe data providers and data consumers. This information is used to establish a run-time environment for processing, which must occur before a connection can be opened and data can be received.

The CDS CD-1.1 provides two transmission modes, unicast and multicast. [Figure 4](#) illustrates differences in data flow between unicast and multicast transmission. Unicast transmission provides direct one-to-one connections between data providers and data consumers where individual data streams are transmitted on each connection. Multicast transmission provides one-to-many connections allowing a data provider to send a single data stream to multiple data consumers at the same time. Data providers and data consumers can operate in both unicast and multicast mode simultaneously.



1. Adapted from Agarwal, October, 2001.

FIGURE 4. UNICAST VERSUS MULTICAST COMMUNICATION

▼ Architectural Design

Traditional Unicast Operation

[Figure 5](#) presents a conceptual view of the *CDS CD-1.1* traditional unicast operation. A provider connects to the *CDS CD-1.1* by attempting a connection to a prescribed host computer. Components of the *CDS CD-1.1* are then invoked to handle the request according to the CD-1.1 protocol. Assuming that the protocol exchanges are correctly executed, the provider proceeds to deliver protocol Data Frames over the connection. A connection remains open indefinitely or until a terminating condition is encountered. If a connection is lost, then the connection may be re-established after a brief wait.

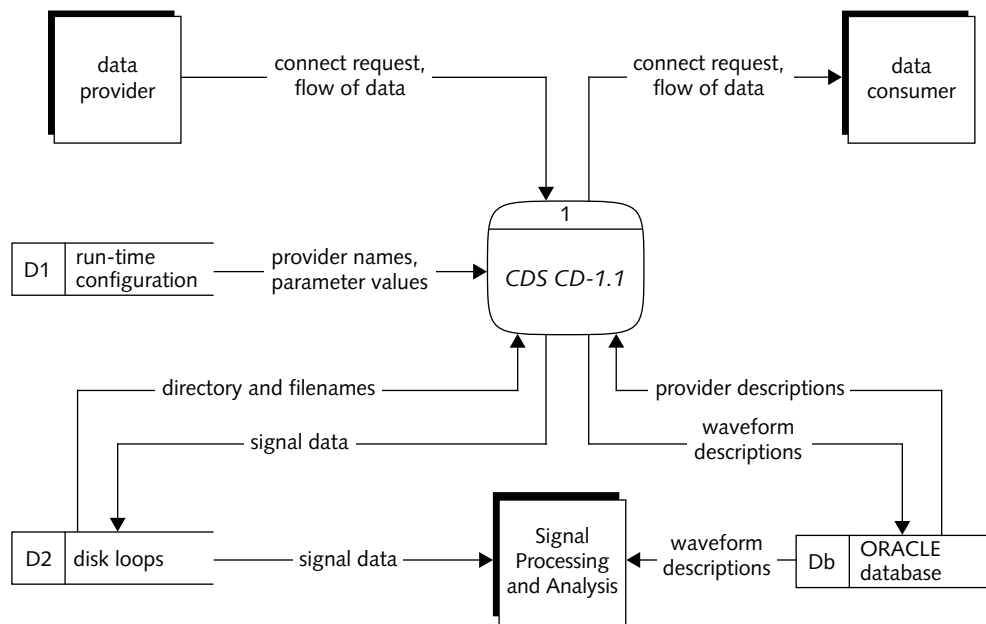


FIGURE 5. OBJECT MODEL OF CDS CD-1.1 UNICAST OPERATION

When Data Frames are received from a provider they are written to disk files that provide a durable store called a Frame Store. Frames are not considered received until a successful store operation has occurred. Acknowledging store operations enables the system to avoid data loss during periods of intermittent communication or interruptions of software operation.

Connections to forwarding destinations are established and managed by *CDS CD-1.1* software. After a forwarding connection is established, the software sends data from the receive Frame Store to the data consumer. Like the connection to data providers, established connections are held open indefinitely. Should a connection terminate, *CDS CD-1.1* automatically attempts to re-establish it.

Multicast Operation

In multicast operation, the conceptual design of the *CDS CD-1.1* is to provide reliable multicast capability allowing a data provider to send a single stream of data to multiple data consumers. Unicast equivalent reliability is achieved through unicast catchup connections whereby a data consumer requests and receives data missed from the multicast transmission. [Figure 6](#) presents a conceptual view of the *CDS CD-1.1* multicast operation.

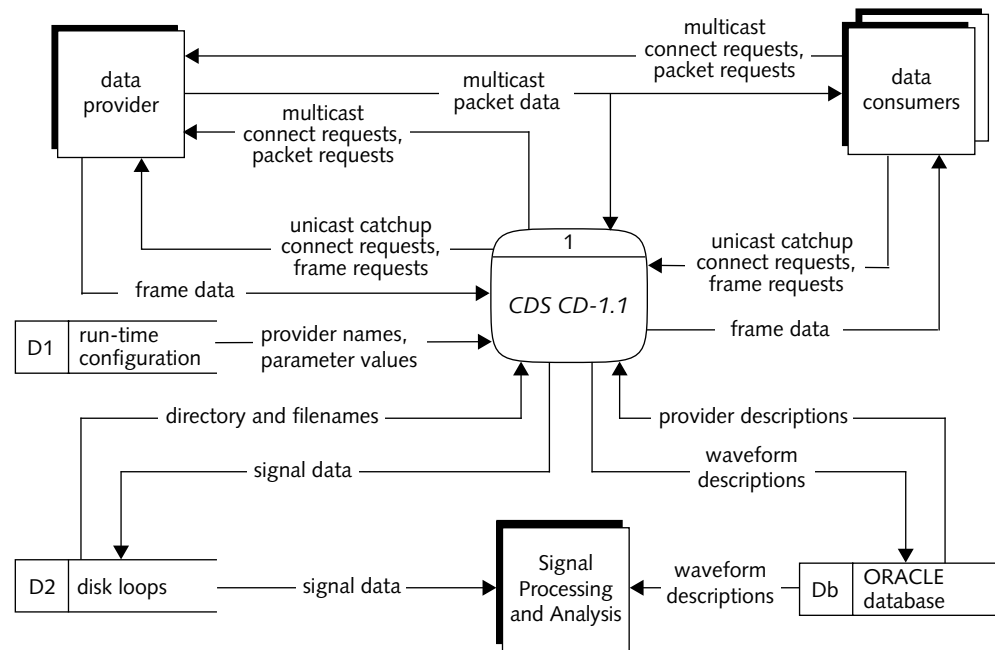


FIGURE 6. OBJECT MODEL OF CDS CD-1.1 MULTICAST OPERATION

▼ Architectural Design

A data consumer requests a multicast connection from a data provider. The provider responds with the multicast connection information. The data consumer joins the group and starts listening to the ongoing multicast transmission. The multicast transmission contains near real time CD-1.1 frames broken into smaller UDP data packets. The data consumer reconstructs CD-1.1 frames from data packets.

Simple multicasting provides best-effort data delivery. *CDS CD-1.1* provides reliable multicasting, or “better” best effort data delivery, through packet-level error detection. Data consumers detect missing packets and send packet retransmission requests back to the station data provider. The data provider collects retransmission requests from data consumers and remulticasts requested packets to the entire multicast group. All data consumers receive reliable multicast service directly from the data provider, including retransmission of missing data packets, meeting most of their data needs

A secondary data transmission mechanism, unicast catchup, works in conjunction with the reliable multicast service to provide highly reliable data delivery equivalent to that of traditional unicast operation. Error detection is performed at the frame level, rather than at the packet level as in the multicast transmission. *CDS CD-1.1* detects missing frames not provided through the multicast transmission and requests a one-to-one unicast catchup connection from the data provider to retrieve the missing frames. After missing frames have been received, the connection is closed.

CDS CD-1.1 allows data consumers to specify a reliability host other than the multicast data provider to service catchup connections. Multiple independent reliability hosts may be established at different locations allowing many more data consumers to be serviced than can be serviced by a single reliability host. Thus, the IDC may be the reliability host for some data consumers, while other reliability hosts support different sets of data consumers.

Data Parsing

Data parsing software extracts data from recently-received frames to produce waveform files and DBMS waveform description (**wfdisc**) records. The received data are made available for signal processing and analysis software by this process.

This processing is independent of frame transmit mode.

DESIGN DECISIONS

The design of *CDS CD-1.1* components takes advantage of existing software libraries, methods, and configurations currently in use by the operational software system. The reuse and commonality of the software aids in both the maintainability and operability of the *CDS CD-1.1*. The design decisions in the following sections pertain to *CDS CD-1.1*.

Programming Language

All of the software components in *CDS CD-1.1* are written in the C and C++ programming languages.

Global Libraries

The software of *CDS CD-1.1* uses the following shared development libraries:

- *libas* digital signature signing and authentication
- *libcancomp* Canadian compression/decompression processing
- *libframelog* specialized logging for frames
- *libgdi* generic DBMS interface
- *liblog* run-time logging software
- *libpar* run-time configuration parameter reading
- *libstdtime* standard time reading and formatting
- *libtable* hash table implementation
- *libwfm* waveform output/writing

▼ Architectural Design

- *libwio* input/output (I/O) of waveform data

CDS CD-1.1 software uses standard code libraries distributed with the C programming language compiler and the UNIX (Solaris) operating system. These libraries are used in their delivered form and are not listed in this document. *CDS CD-1.1* software is also linked to the following public domain library for use in authentication processing:

- *openssl* secure socket layer software

Database

CDS CD-1.1 software uses an ORACLE 8i DBMS to aid in the management of connection information and to store descriptions of waveform data parsed from received frames. All database queries are affected through the shared library *libgdi*. This library isolates the software from explicit details of DBMS interaction by providing a generic front-end.

Interprocess Communication (IPC)

The design of the *CDS CD-1.1* software components attempts to decouple processes from one another. As a result, direct interprocess communication is limited. The following paragraphs identify the communication methods that are used.

UNIX socket interprocess communication (IPC) is used as the communication method for transporting CD-1.1 protocol frames between data providers and data consumers. Additionally, sockets are used by *Connection Manager* to distribute inbound connections to *Connection Manager Servers* on the LAN at the data center.

UNIX pipes are used for communication between *Frame Exchange* and *Exchange Controller* components of the *CDS CD-1.1*, which always execute on the same host processor. The IPC pipes between these components are used to pass “frame messages” for coordinating frames sent and acknowledging those received. In the event of a terminating condition the pipes are also used to signal a graceful shutdown.

UNIX data streams are used to provide information from a child process to the parent process. This method is used with the system-allocated standard out (`stdout`) and standard error (`stderr`) streams for processes that execute on a single host computer and that have a parent-child relationship.

A Frame Store is a repository for saving protocol frames and is used as an interface medium for sharing data between processes. Processes gain access to a Frame Store through Application Program Interface (API) calls in the Frame Store software library.

The DBMS and file system are used as an interface for providing waveform data and waveform descriptions to other IDC applications. This method of providing an interface to these data was employed by previous generations of CDS software. The *CDS CD-1.1* complies with existing definitions and methodologies so as to not influence the processing of other system elements.

File System

The file system for *CDS CD-1.1* host computers has directory hierarchies with space for parameter files, run-time data stores, and process logs. All of the *CDS CD-1.1* processes take run-time configuration parameters and environment variables as input. Values for these inputs are provided in a series of parameter files, which are contained in the file system. Configuration information is usually stored in a common location for some, if not all, of the *CDS CD-1.1* processes. If a common location is used, the file system with the parameter files must be Network File System (NFS) mounted by each of the *CDS CD-1.1* host computers.

Frame Stores are implemented as directories and files in the file system. Frame Stores are used by *CDS CD-1.1* software as an interface mechanism as well as a place for retaining processed data and audit information. The file systems that house Frame Stores are usually NFS mounted by a number of host computers. NFS mounting distributes the *CDS CD-1.1* processing load over multiple computers.

▼ Architectural Design

Each process of the *CDS CD-1.1* creates a log file for writing messages related to process execution. Log files are American Standard Code for Information Interchange (ASCII) text files, which in normal circumstances are accessed relatively infrequently. A log file is best written to a file system resident on the host computer.

Traditional Unicast Design Model

The *CDS CD-1.1* software suite is designed to provide an auditable, robust system for delivering, parsing, and forwarding time-series data using unicast transmission technology. As the data receiving front-end of the IDC, the *CDS CD-1.1* is efficient and responsive enough to keep up with the flow of data from hundreds of sources. *CDS CD-1.1* software also reacts to failures in communications with peers in remote geographic areas. The design is noteworthy in two regards: a transactional paradigm is employed, and independent components are responsible for various processing activities.

The transactional model provides positive feedback to protocol participants regarding frame deliveries and facilitates the ability to audit protocol activity with a given protocol peer. With positive feedback, the progress of data delivery is declared only after it has occurred. For example, an acknowledgement for a frame receipt is not generated until after the frame is successfully saved to the Frame Store. This also supports data recovery from unexpected communication or application software terminations.

The *CDS CD-1.1* design contains independent processing components for implementing various *CDS CD-1.1* capabilities. This design enhances the robustness of the system by distributing needed processing among processing components. In such a design, if a particular component unexpectedly terminates, other capabilities of *CDS CD-1.1* continue to be provided in a relatively unaffected fashion. Distributing the processing capabilities over several hosts also allows resource-intensive processing to be segregated, which increases system throughput and responsiveness (resources include I/O processing, computation, and DBMS transactions).

Multicast Design Model

The *CDS CD-1.1* expands on the traditional unicast design to provide a highly reliable multicast service that provides the same auditability as traditional unicast. Current multicast technologies based on UDP are inherently less reliable than unicast technologies based on TCP/IP. However, the promise of significant network bandwidth savings makes multicasting attractive. The design allows a data provider to support more data consumers directly for most of their data needs without dependence on an intermediate forwarding facility, such as the IDC. The *CDS CD-1.1* multicast design addresses the limitations of multicast technology while maintaining the advantages of providing a multicast service with reliability equivalent to that of the *CDS CD-1.1* traditional unicast design.

Custom Multicast Solution

A custom implementation was chosen rather than a COTS solution for CD-1.1 multicasting. State-of-the-art reliable multicasting COTS systems only provide better best-effort data delivery. Full reliability is achieved with such systems by adding higher level application-specific error handling. Further, these technologies are still emerging and industry-wide standards for protocols and interfaces have not yet been established. As a custom application was needed in any case, it was simpler to design a full custom system rather than relying on immature technologies that do not provide a full solution [\[SAIC-01/3068\]](#).

Data Provider Provides Reliable Multicasting

A primary motivation for using multicast technology is to allow a data provider at a station to directly service the needs of most data consumers without dependence on an intermediate forwarding facility, such as the IDC. The design of the multicasting subsystem provides this by allowing all data consumers to send requests for retransmission of missing data packets directly to the data provider. Requests are sent in the form of Packet Negative Acknowledgements (PNacks) via unicast UDP as shown in [Figure 7](#). The data provider collects PNacks, removes duplicate requests and remulticasts requested data packets to the entire multicast group.

▼ Architectural Design

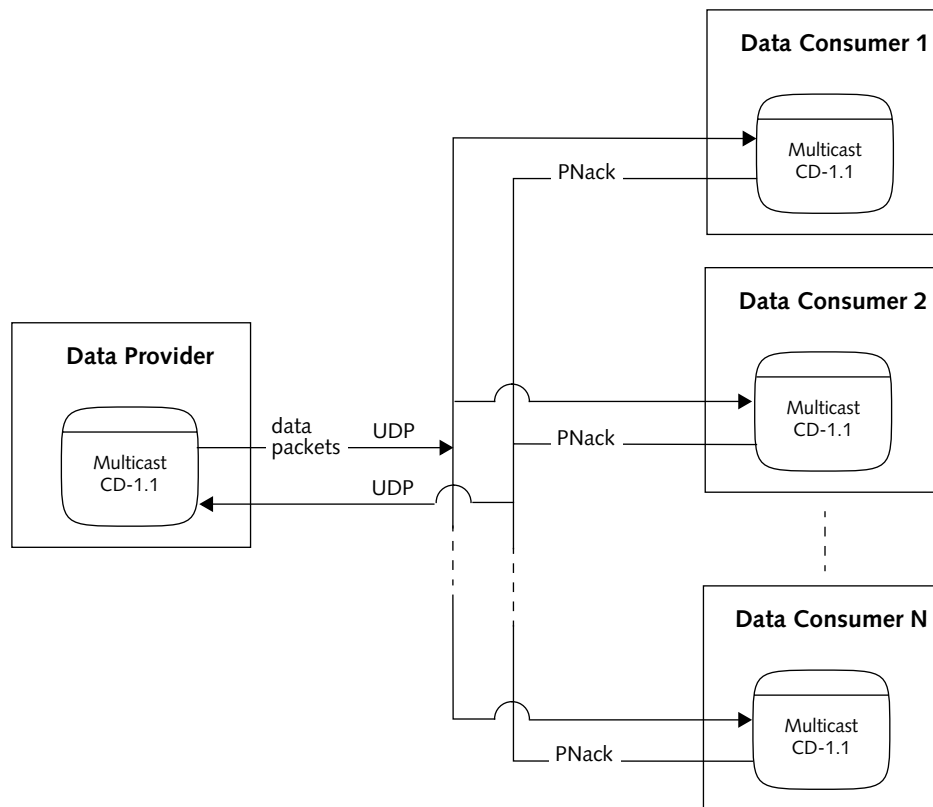


FIGURE 7. RELIABLE MULTICAST SUBSYSTEM

Unicast Catchup

Multicasting is inherently a best-effort transport technology. Reliable multicasting, i.e. multicasting with retransmission requests, only provides a “better” best-effort solution. Reliable multicasting cannot service retransmission requests for more than a short period before it must give up and continue to provide multicasting of near real-time data. To achieve full CD-1.1-level reliability, a unicast catchup mechanism is used to deliver data that fails to be delivered via multicasting. The unicast catchup mechanism allows a data consumer to make a one-to-one TCP/IP connection with a data provider to request missing data. This also provides a catchup mechanism after outages of either the data provider or data consumer.

Separate Multicast and Unicast Catchup Subsystems

Separate subsystems are used for the multicasting and unicast catchup data delivery as shown in [Figure 8](#). Separate subsystems provide several benefits. Independent subsystems follow the design model established in the traditional unicast design of multiple independent components that contribute to the robustness of the system. Each component is relatively simple and concentrates on a single part of the job. Components can be distributed over multiple hosts. An unexpected termination and restart of a single component can occur without disruption of other components.

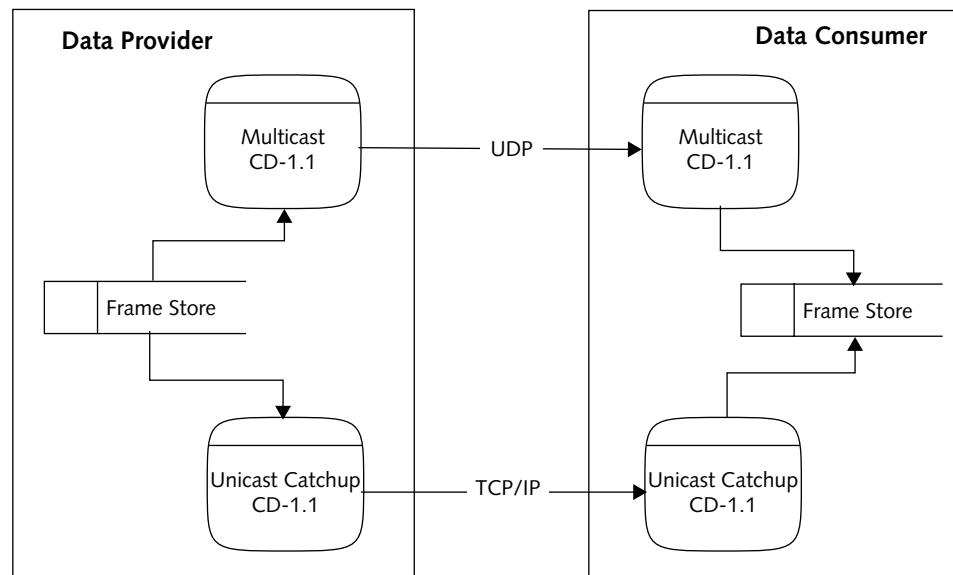


FIGURE 8. SEPARATE MULTICAST AND UNICAST CATCHUP SUBSYSTEMS

Full CD-1.1 reliability can only be achieved with CD-1.1 frame-level error handling. This capability is already provided by the unicast components, a robust and proven design. With only minor modifications, the existing traditional unicast com-

▼ Architectural Design

ponents allow a data consumer to request a unicast catchup connection and request any missing data still stored in the Frame Store of the data provider. Reuse of proven components reduces implementation risk and cost.

Reliability Hosts for Unicast Catchup Subsystem

A data provider at a station may support multiple unicast catchup connections from data consumers. However, bandwidth of the WAN connection, processing power and storage capacity of the data provider provide practical constraints on the number of data consumers that can be allowed to connect directly to a station data provider. Separate reliability hosts, with higher network bandwidth, processing power, and storage capacity, can support many more unicast catchup connections to data consumers than can be supported directly by a station data provider. [Figure 9](#) shows the concept of operation for the unicast catchup subsystem using reliability hosts.

Any host that services unicast catchup connections is a reliability host. The station data provider is a reliability host for one or more data consumers and secondary reliability hosts. The IDC and other facilities with sufficient network bandwidth and resources to support unicast catchup connections operate as secondary reliability hosts. The secondary reliability hosts request unicast catchup connections from the station data provider to retrieve frames not received from the multicast transmission. Other data consumers request unicast connections from the secondary reliability hosts and request frames from the local Frame Store. The unicast catchup connections between the secondary reliability hosts and station data providers and the unicast catchup connections between other data consumers and the secondary reliability hosts are completely independent from each other.

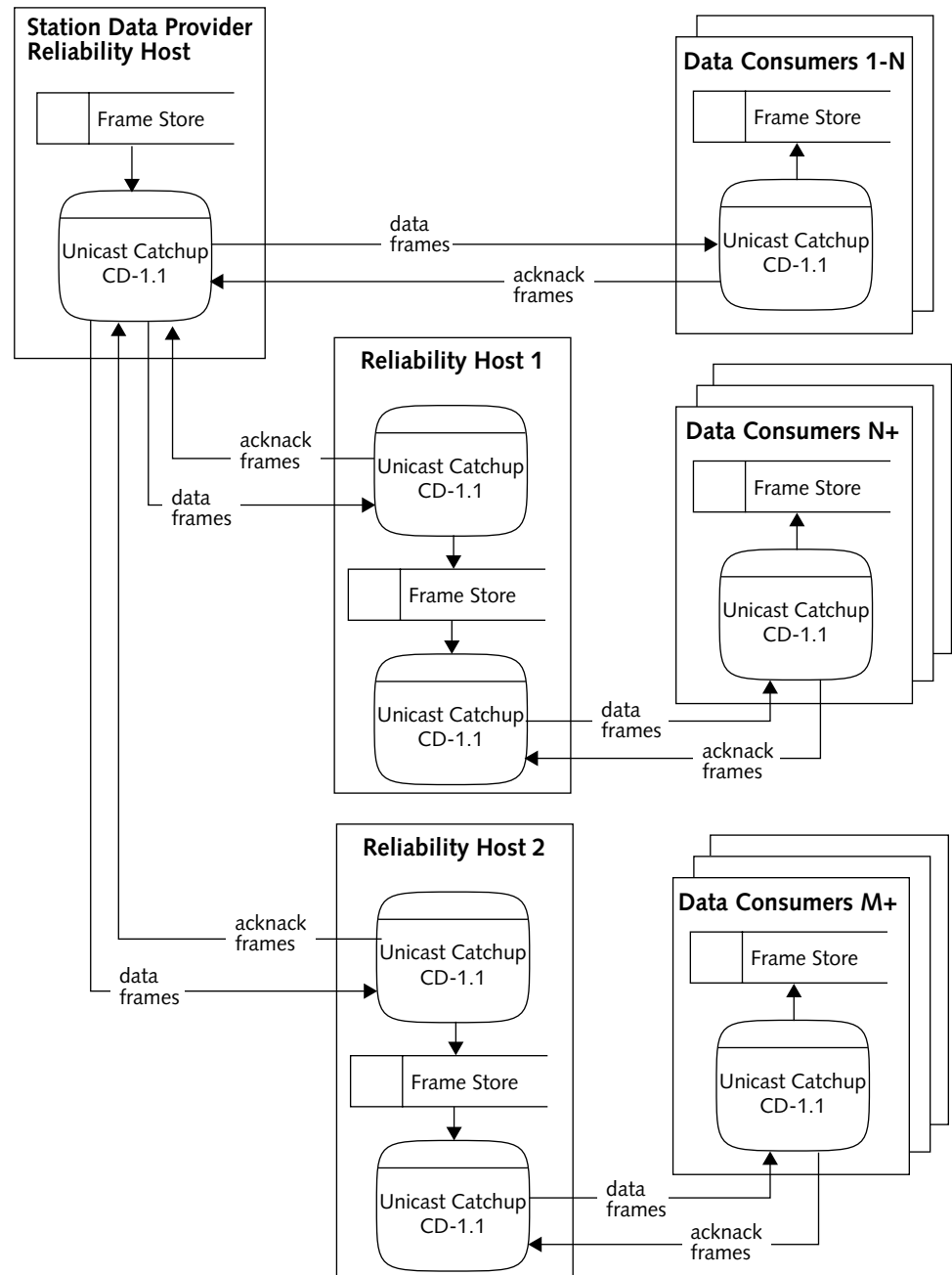


FIGURE 9. RELIABILITY HOSTS FOR UNICAST CATCHUP SUBSYSTEM

▼ Architectural Design

Multicast Connection Initiation

Unlike TCP/IP, UDP is a connectionless protocol. The multicast data provider transmits data without any direct connection to a data consumer. The data provider simply sends data to the multicast group's IP address. If a data consumer joins the multicast group it should receive the data. To join a multicast group, a data consumer requests a "connection" to the data provider in the form of a Connection Request Frame and in return receives a Connection Response Frame containing the IP address and port number of the multicast group and IP address and port number of the PNack receiver (Figure 10). The data consumer joins the multicast group and starts listening to the data stream. The Connection Response Frame is not provided to the data consumer until its connection request has been verified.

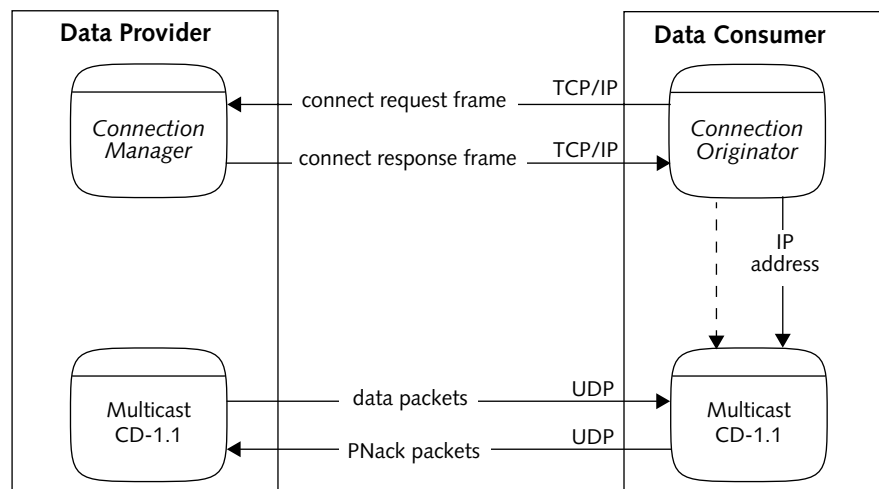


FIGURE 10. MULTICAST CONNECTION INITIATION

Unicast Catchup Connection Initiation

Unicast catchup connection initiation is the same as for traditional unicast connections except that the data consumer initiates the connection (Figure 11). In normal operation with good multicast transmission, no unicast catchup connection is

needed. If a data consumer is missing data for any reason, it requests a unicast catchup connection by sending a Connection Request Frame to the data provider. The data provider responds with a Connection Response Frame. The data consumer then sends an Option Request Frame containing the list of missing frame sequence numbers and the data provider responds with an Option Response Frame. Once the connection has been established, the data provider begins sending requested frames that are available in the Frame Store. After all the requested frames have been sent and positively acknowledged, the data provider terminates the connection.

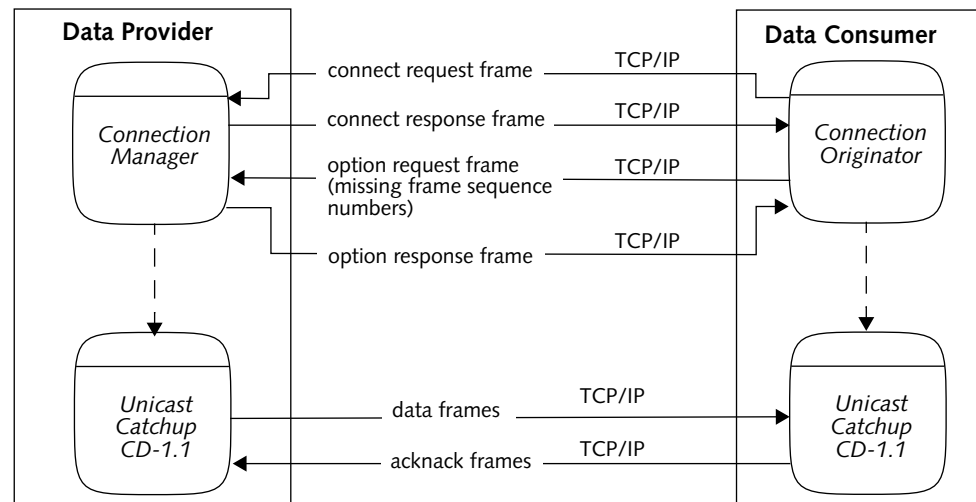


FIGURE 11. UNICAST CATCHUP CONNECTION INITIATION

Multicast Startup Time

CD-1.1 multicast is intended to provide near real-time data to data consumers. It is not intended to back fill long periods of missing data after an outage. At startup, a data provider begins sending near real-time data minus a small lookback period. A data consumer may join a multicast group at any time and start receiving current data but may not request that the data provider remulticast catchup data. That functionality is provided by the unicast catchup subsystem.

▼ Architectural Design

The small lookback period at startup allows the multicast subsystem to recover from a short outage, such as a restart of the data provider, without causing a data consumer to use a unicast catchup connection to fill in missing data.

Database Schema Overview

CDS CD-1.1 software uses the ORACLE database for the following purposes:

- to verify connection requesters
- to determine the *CDS CD-1.1* host for servicing a data provider
- to determine the site sensor parameters for parsing data
- to determine the location and form of disk loop files for storing parsed waveform data
- to store descriptions of time-series data written to disk loop files

[Table 1](#) shows the database tables used by the *CDS CD-1.1*. The name field identifies the database table. The mode field is “R” if *CDS CD-1.1* reads from the table and “W” if it writes to the table.

TABLE 1: DATABASE TABLES USED BY CDS CD-1.1

| Name | Mode | Description |
|--------------------|------|---|
| affiliation | R | specifies the affiliation between sites and reporting stations, that is, which sites are reported by which stations |
| alphasite | R | specifies data providers whose connections are accepted, and identifies the dlman to handle the connection |
| dlman | R | (<i>Disk Loop Manager</i>) identifies computers capable of hosting a <i>CDS CD-1.1</i> connection |
| sensor | R | provides descriptive information about sensors in the IMS (the sensor instruments at the sites) |
| site | R | provides descriptive information about a given reporting site (data provider) |
| sitechan | R | provides descriptive information about the data channels at a given site (data provider) |

TABLE 1: DATABASE TABLES USED BY CDS CD-1.1 (CONTINUED)

| Name | Mode | Description |
|----------------|------|--|
| wfconv | R | provides information used to convert data from a specific site from its native format into CSS 3.0 format (waveform files) |
| wfdisc | W | provides waveform description records for data parsed into the waveform files |
| wfproto | R | provides prototype information used for the production of wfdisc records |

FUNCTIONAL DESCRIPTION

The CDS CD-1.1 is composed of the following high-level components:

- *Connection Manager* manages data connection requests
- *Data Center Manager* manages process execution at a data center
- *Connection Originator* requests data connections
- *Exchange Controller* provides policy logic for *Frame Exchange*
- *Frame Exchange* handles transmission of protocol frames
- *Multicast Sender* deconstructs protocol frames into data packets and sends to data consumers via multicast
- *Multicast Receiver* receives multicast data packets and reconstructs protocol frames
- *Data Parser* parses protocol Data Frames into a format usable by data processing components
- *Frame Store Stager* manages staging of Frame Store files for archiving
- *Authentication* provides DSA authentication capabilities

Traditional Unicast Functional Description

[Figure 12](#) presents the data flows of CDS CD-1.1 software traditional unicast mode at the IDC and shows three important functions: receiving data, forwarding data, and providing data for processing. Receipt of data from an IMS station is pictured on the left side of [Figure 12](#); forwarding data to an NDC (or other recipient) is pictured on the right side of the figure, and parsing data is pictured at the bottom of the figure.

Flow of inbound protocol frames begins when a connection request from an IMS station *Connection Originator* is accepted by the IDC *Connection Manager* (process 1). After authenticating the request (process 2), control is passed to the *Exchange Controller* (process 3), which controls the exchange of frames through the IDC *Frame Exchange* (process 4). The IDC *Frame Exchange* is connected to both the *Exchange Controller* (process 3) and the IMS *Frame Exchange* at the data source. This interface with the IMS *Frame Exchange* is used to provide acknowledgements and commands to the IMS station. Received frames are stored in a Frame Store (D1), whose contents are indexed in a frame log imbedded within the Frame Store.

To forward data to an NDC, a *Connection Originator* at the IDC (process 6) exchanges protocol frames with the NDC *Connection Manager* to set up an interface between a *Frame Exchange* at the IDC (process 8) and a *Frame Exchange* at the NDC. The IDC *Frame Exchange* sends data to the NDC via this interface. Control is passed to an IDC *Exchange Controller* (process 7) to manage this connection.

Data Frames are parsed by *Data Parser* (process 5), and the result is written to disk loops in CSS 3.0 format for use by other IDC software.

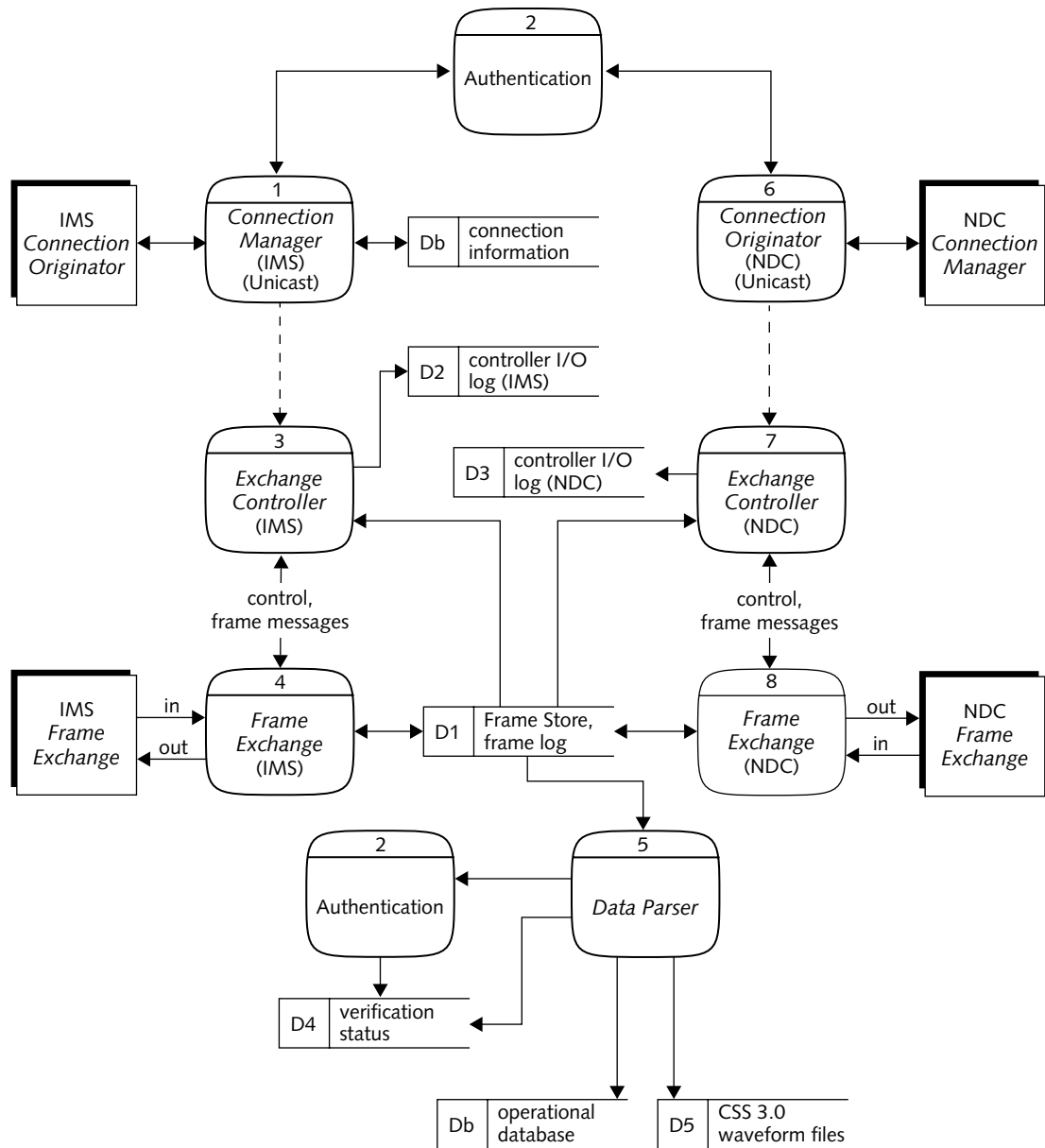


FIGURE 12. FUNCTIONAL DESIGN OF CDS CD-1.1 TRADITIONAL UNICAST OPERATION

Multicast Functional Description

[Figure 13](#) shows the functional design of CD-1.1 software multicast mode at the IDC. The four functions depicted are receiving multicast data, requesting and receiving missing data via unicast catchup connections, servicing requests for data to other data consumers via unicast catchup connections, and providing data for processing. Receipt of data from an IMS station is pictured on the left side of the figure, transmission of data to an NDC (or other data consumer) is pictured at the upper right, and parsing of data is pictured at the lower right of the figure.

Reception of inbound multicast frames begins when an IDC *Connection Originator* (process 1) sends a multicast connection request to the IMS *Connection Manager* which sends a connection response. Once the connection response is received, control is passed to *Multicast Receiver* (process 2), which listens to the multicast stream of data packets, reconstructs protocol frames and writes them to the Frame Store. *Multicast Receiver* detects missing packets and sends PNack packets back to the IMS station. If the packets are still available in internal memory, *Multicast Sender* at the IMS station resends the packets via the multicast stream.

A unicast catchup connection to the IMS station is initiated when missing frames are detected in the local Frame Store. The *Connection Originator* (process 5) sends requests a connection from the *Connection Manager* at the IMS station. When the connection is accepted, *Connection Originator* passes control to *Exchange Controller* (process 4) that interacts with *Frame Exchange* (process 3) to receive missing frames from the *Frame Exchange* at the IMS station.

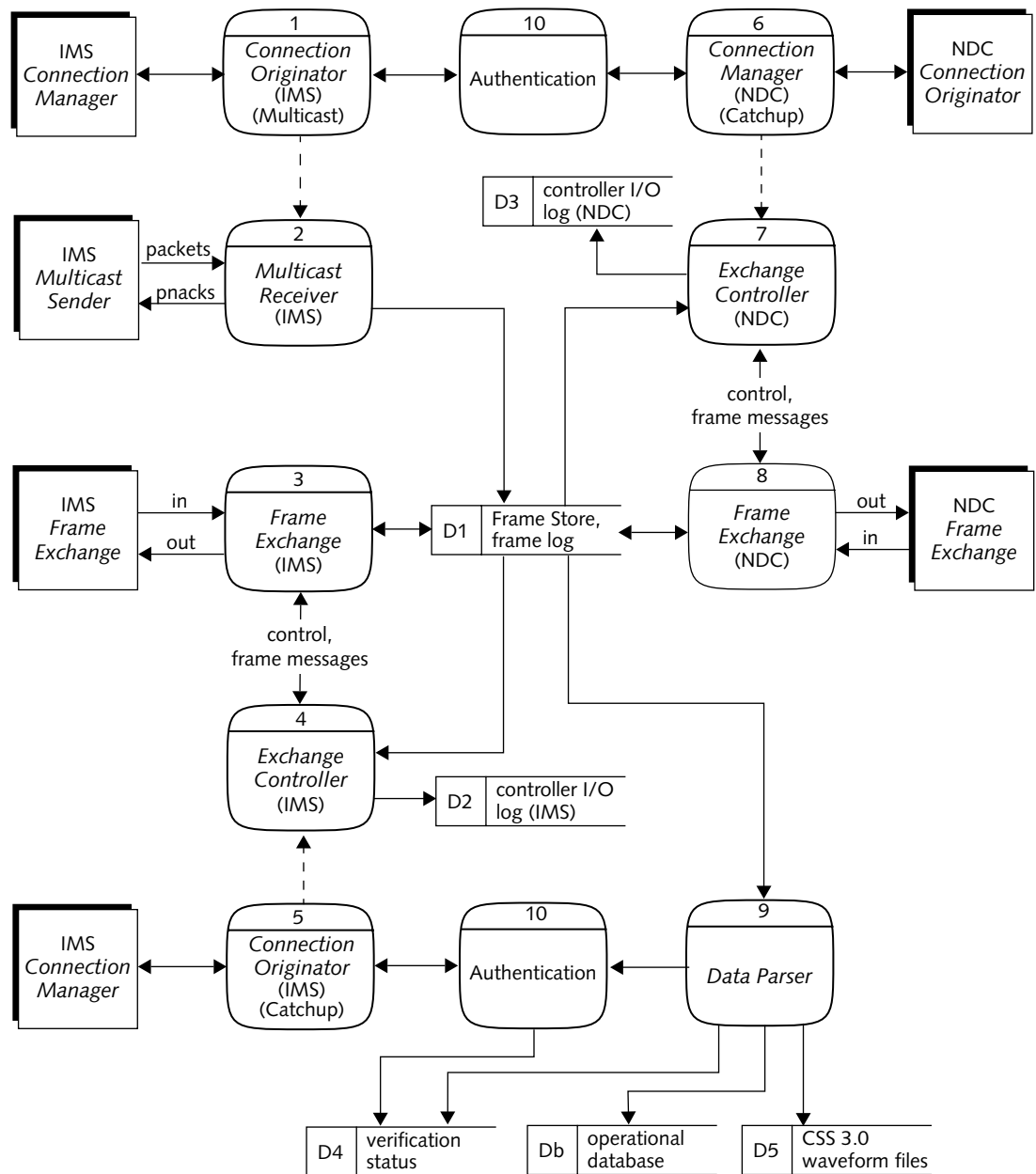


FIGURE 13. FUNCTIONAL DESIGN OF CDS CD-1.1 MULTICAST OPERATION

▼ Architectural Design

Similarly, other data consumers may request unicast catchup connections from the IDC to request missing protocol frames. The connection is initiated by a *Connection Originator* at the remote data consumer. *Connection Manager* (process 6) responds to a connection request and initiates an *Exchange Controller* (process 7) and *Frame Exchange* (process 8) pair to service the connection. After the requested data is sent and acknowledged, *Frame Exchange* terminates the connection and exits. The unicast catchup operation only responds to requests for missing data; it does not forward all data as in traditional unicast operation.

Data Frames are parsed by *Data Parser* (process 9), and results are written to disk loops in CSS 3.0 format for use by other IDC software.

SOFTWARE COMPONENTS

The following sections describe *CDS CD-1.1* software components. Components in this context may or may not map directly to a computer process. Components are entities that provide a significant processing capability to the overall *CDS CD-1.1*.

Connection Manager

Connection Manager, in concert with *Connection Originator*, establishes connections between a data provider and a data consumer. *Connection Manager* maintains the addressing information and connection policy of *CDS CD-1.1* as defined in [\[IDC3.4.3Rev0.2\]](#) for the three connection types, traditional unicast, multicast, and unicast catchup.

Traditional unicast connections are requested by *Connection Originator* at a data provider. *Connection Manager* exchanges connection information with *Connection Originator* and establishes a TCP/IP connection. After the connection is established, *Frame Exchanges* communicate over the private link without any further assistance from *Connection Manager*.

For multicast connections, *Multicast Sender* at the station is continuously sending data to the multicast group regardless of whether a data consumer is listening. *Connection Manager* at a station data provider responds to a connection request

from *Connection Originator* at a data consumer by sending connection information for the multicast group and PNack UDP connections. *Connection Manager* at the station then exits. There is no direct interface between *Connection Manager* at a station data provider and *Multicast Sender*.

Unicast catchup connections are established much the same way as traditional unicast connections however the connections are requested by *Connection Originator* at a data consumer rather a data provider. *Connection Manager* on the data provider or reliability host exchanges connection information and establishes a TCP/IP connection to be used by the *Frame Exchanges*.

For traditional unicast and unicast catchup connections, *Connection Manager* interfaces with *Exchange Controller* through process inheritance. After establishing a TCP/IP connection, *Connection Manager* execs *Exchange Controller*. *Exchange Controller* inherits the connection information (sockets and identity of the other end) from *Connection Manager*.

Connection Manager at a data center uses the database to validate connection requests by looking up acceptable connection sites and their addresses. *Connection Manager* interfaces with UNIX via *inetd*(1) and takes advantage of UNIX services for port monitoring and invocation.

Data Center Manager

Data Center Manager provides process control for several CDS CD-1.1 processes that are executed at a data center. This process control is provided for those processes that require monitoring and stimulation/instantiation from within the data center, in contrast to processes that are instantiated by stimulation from outside of the data center. An example of an externally stimulated process is the process that services inbound data from an IMS station. An example of an internally instantiated process is *Data Parser*.

▼ Architectural Design

Data Center Manager instantiates child processes at a data center. The child processes are monitored by *Data Center Manager* to detect exit conditions and configured output. When a child process exits, the manager has the capability to react to the condition via rules defined explicitly for the child. Reaction can include restarting the same process, starting a different process, or doing nothing.

Data Center Manager provides execution startup commands in the form of a UNIX command line with optional command-line arguments. *Data Center Manager* also interfaces with managed processes via UNIX `stdout` and `stderr` data streams. These streams are monitored by *Data Center Manager* to log process status. If a child process provides configured output to *Data Center Manager*, the *Data Center Manager* has the ability to treat the output as a command, write the output to a log file, or ignore the output.

An interface to the operating system exists to issue signals to managed processes. The interface to the operating system is also used for obtaining the process IDs of managed processes and for time.

Connection Originator

Connection Originator initiates a CD-1.1 connection. *Connection Originator* is responsible for establishing a TCP/IP connection and providing CD-1.1 protocol transactions that result in an agreement to maintain a connection. *Connection Originator's* frame traffic is limited to that required for establishing a communications path. After a communication path is established, *Connection Originator* spawns a child process to continue protocol exchanges. *Exchange Controller* is used for traditional unicast and unicast catchup connections and inherits the TCP connection established by *Connection Originator*. *Multicast Receiver* is used for UDP multicast connections. It makes no use of the TCP connection.

A command-line interface is used to provide run time configuration information to *Connection Originator*. *Connection Originator* interfaces with the UNIX operating system via system calls to attach to socket file descriptors and to obtain information about connected computers. *Connection Originator* is dependent on the cor-

rect configuration of the operating system with respect to IP addresses and computer names and provides no capability to manipulate host computer network information.

Connection Originator interfaces with a protocol peer computer to obtain the remote socket assignment (connection request and response) and to exchange connection information. A *Connection Manager* process on the protocol peer monitors and manages port assignments, and provides assignments to *Connection Originator*.

Connection Originator interfaces with *Exchange Controller* and *Multicast Receiver* through process inheritance. The interface to the child process is unidirectional and involves a single transaction. *Connection Originator* is responsible for providing the needed startup information to the child. After it is `exec'd`, *Connection Originator* becomes the child process.

Connection Originator has an interface to a logging capability. This interface is used to log connections and attempted connections.

Exchange Controller

Exchange Controller implements the policy of how to act upon protocol frames. *Exchange Controller* performs simple distribution and prioritization based on the frame source and the frame type for both inbound and outbound frames. *Exchange Controller* encapsulates the ordering policy for outbound frames and provides notifications for inbound/received frames.

Exchange Controller's interface with *Frame Exchange* is through a UNIX pipe. *Exchange Controller* sends frame messages on the outbound pipe and receives frame messages on the inbound pipe.

Exchange Controller may interface with other components of the system indirectly through a Frame Store. *Exchange Controller* monitors a Frame Store for newly received frames and acts on frames discovered. For example, in a data forwarding

▼ Architectural Design

configuration, newly received Data Frames are discovered by a forwarding instance of *Exchange Controller* and communicated to *Frame Exchange* for the purpose of forwarding.

Frame Exchange

The primary function of *Frame Exchange* is to reliably transport frames from one Frame Store to another. Each *Frame Exchange* has a single associated *Exchange Controller*, which directs the operation of *Frame Exchange* using frame messages sent over a pipe. Each *Frame Exchange* is connected by a TCP/IP socket to a corresponding *Frame Exchange* at another site to which it sends and from which it receives frames.

Frame Exchange has two interfaces: a private pipe to its associated *Exchange Controller* and a socket connection to a *Frame Exchange* on another node. Communication with *Exchange Controller* is done by frame messages. Frame messages handle the full interface functionality required between *Frame Exchange* and its *Exchange Controller*.

The socket interface is used only to send and receive frames. The syntax of frames is described fully in [\[IDC3.4.3Rev0.2\]](#). Except for AckNack messages, which are handled directly by *Frame Exchange*, all frames are stored in an appropriate frame set within a Frame Store, and notification is sent to *Exchange Controller*.

Frame Exchange interfaces with the Frame Store to store received frames and to send retrieved frames.

Multicast Sender

Multicast Sender reads CD-1.1 protocol frames from a Frame Store, breaks them down into UDP packets and sends them via IP multicast. An internal buffer is maintained containing transmitted packets. PNack packets identifying missing packets are received from data consumers via a separate UDP port. If requested packets are still in the internal buffer, *Multicast Sender* remulticasts them to all data

consumers. *Multicast Sender* waits a configurable hold off period before accepting subsequent PNacks for the same packets to allow for network round trip transmission time.

The size of the internal packet buffer is configurable. However, it is significantly smaller than a Frame Store, on the order of ten minutes compared to a Frame Store, which is typically seven days. The packet buffer is stored in volatile memory and is lost in a processor restart. At startup, *Multicast Sender* reads data from the Frame Store beginning with frames created at the current time minus a small configurable lookback period.

When the internal buffer is full, oldest data is overwritten by newest data. Once data are overwritten, they can no longer be transmitted via multicast so *Multicast Sender* cannot respond to PNAck requests for that data. Once started, *Multicast Sender* continues to send data packets to the multicast group whether or not any data consumers are listening.

The upper rate of multicast transmission is configurable. UDP has no native rate control and, if unconstrained, could overwhelm the network. The transmission rate must be set low enough to prevent congestion yet sufficiently high to keep up with current data plus a tolerable amount of retransmissions due to PNAck requests.

Multicast Receiver

Multicast Receiver is responsible for receiving UDP data packets, reconstructing CD-1.1 protocol frames and writing them to a Frame Store. Data packets are received through a socket connection to the IP multicast group and stored in an internal buffer. Missing packets are requested by sending a PNAck packet to the data provider via a unicast UDP connection. The data provider responds by remulticasting missing packets to the entire group.

After all data packets for a protocol frame are received, they are reconstructed into a frame and written to a Frame Store. If all packets for a frame are not received before the internal circular buffer rolls over, the frame is abandoned and can no longer be received through multicast. The size of the internal buffer is configurable but should match the size of the buffer on the data provider.

▼ Architectural Design

Multicast Receiver is initiated by *Connection Originator* after a multicast connection is established. *Connection Originator* provides *Multicast Receiver* to the IP multicast group and UDP PNack receiver.

Data Parser

Data Parser converts CD-1.1 protocol time-series data to the CSS 3.0 format readable by automatic and interactive processing software.

Data Parser polls frame sets at a data center for newly arrived Data Frames. New frames are parsed to extract station and channel names, duration, data time, and compression format. Time-series data are converted to CSS 3.0 format and written to disk files in a disk loop. A disk loop is a collection of files logically ordered in a time-series loop. *Data Parser* places time-series data in their proper position in disk loop files regardless of the time order in which the data arrive and are processed. After data are written to the file system their location is referenced in the **wfdisc** table of the DBMS. The data become visible to other IDC applications through the DBMS references.

Data Parser is a stand-alone capability and uses a command-line interface to provide run-time configuration information.

The DBMS interface is used primarily to record information about the time-series data placed in disk loops, that is, station, channel, time, and disk-loop location information (waveform description—**wfdisc** data). Time-series descriptive data in the database are used by signal processing and analysis for accessing received data. Secondly, the DBMS interface provides and stores processing state information. This information is particularly useful at startup time, for example, the status of polling of a frame set.

Data Parser interfaces with a Frame Store to retrieve inbound CD-1.1 frames. Within Frame Store *Data Parser* accesses one frame set for each station providing CD-1.1 protocol data. Consequentially, a given *Data Parser* may poll and process data from a number of frame sets. Multiple frame set activities are also reflected in the interface to the DBMS in that state, because each set must be recorded.

Data Parser uses an interface to the UNIX file management system to manage disk loop files and to store data. The number and size of the files are configurable for optimum performance, utility, and supportability.

Frame Store Stager

Frame Store Stager is the interface between the CDS CD-1.1 and the Archiving Subsystem. *Frame Store Stager* is flexible enough to support several existing archiving systems. *Frame Store Stager* only needs to run periodically; *cron*(1) is a suitable mechanism for an operational implementation.

Frame Store Stager moves the oldest Frame Store files out of the active Frame Store and into a staging area. After they are moved, processes interacting with the active Frame Store are no longer able to access the files. The staging area is a logical construct, which means the files are disassociated from the active Frame Store. The files might not move (a configuration option), but the active Index no longer references the files.

Frame Store Stager identifies the archive-ready files to the Archive Subsystem by way of database table entries. The Archive Subsystem periodically examines the database and handles (copy/confirm/delete) the files that are waiting to be archived. The Archive Subsystem has the responsibility of deleting files from disk after their transfer to the archive medium is confirmed.

The primary purpose for archiving the Frame Store is to allow re-authentication of data. The original frame must be saved intact to meet this requirement. *Frame Store Stager* could serve as an archive of waveform data; however, this method requires a data parsing step to extract the data.

Authentication

Authentication provides the ability to apply and authenticate DSA signatures in CD-1.1 frames. Each frame as well as each Channel Subframe in the CD-1.1 protocol contains authentication data fields.

▼ Architectural Design

When a frame is created the creator optionally sets and fills authentication data fields for carrying the digital signature of the creator. Authentication supports the specification of a key identifier used in generating the signature. Given the key ID and the frame to be signed, the processing fills the frame signature field and sets authentication data values to represent the presence and size of the authentication data. Applying an authentication signature does not result in the encryption of the frame's content.

Signature authentication is also provided to verify the authenticity of CD-1.1 frames. This processing capability validates the digital signatures, where these signatures are evaluated within the context of the accompanying frame or subframe.

Authentication capability is provided as an application program interface to a software library. Through this interface any *CDS CD-1.1* component may invoke signature authentication services. Requesters provide a frame or Channel Subframe to be signed or authenticated and receive the results of that request.

The signature authentication capability has a read-only interface with a Frame Store to obtain frames that are evaluated for signature validity.

The results of frame and Channel Subframe authentication processing are recorded using an interface to the frame logs. The use of this interface is activated via initialization configuration, such that the requester may assume responsibility for recording authentication results.

INTERFACE DESIGN

This section describes *CDS CD-1.1* interfaces with other IDC systems, external users, and operators.

Interface with Other IDC Systems

CDS CD-1.1 has no direct interface with the other processes in the IDC system. However, *CDS CD-1.1* does indirectly interface with all other system elements that process signal data. *CDS CD-1.1* receives time-series data and parses them into disk

loop waveform files, which are described by database waveform description (**wfdisc**) records. The interface to other system elements is through these database records and disk-loop files.

Interface with External Users

CDS CD-1.1 interfaces with external users to receive data from data providers and to forward data to data consumers. These interfaces are conducted through UNIX sockets. The IP stack is used over these sockets, employing TCP/IP (or UDP for multicast) for the low-level transportation mechanism. The CD-1.1 protocol is at the application layer of the protocol, as documented in [\[IDC3.4.3Rev0.2\]](#).

Interface with Operators

The operational processing of the *CDS CD-1.1* is designed to be automated. In other words, no operator action is needed to respond to data connection requests, to invoke data parsing, or to forward data after the software suite is configured and started. With this design approach the following interfaces result:

- A command-line interface is used to start *Data Center Manager* processing on each host selected for either parsing or forwarding data. This interface permits specification of a run-time configuration file and a process name.
- Each process of the *CDS CD-1.1* writes significant processing events to a log. These log files are ASCII text and are viewable by any operator. These files are generally sparse and contain information that may be of greater interest to a software engineer than a system operator. However, error conditions are recorded in these files, and when possible all terminating failures are logged.

Chapter 3: Detailed Design

This chapter describes the detailed design of the *CDS CD-1.1* and includes the following topics:

- [Data Flow Model](#)
- [Processing Units](#)
- [Multicast Protocol](#)
- [Database Description](#)

Chapter 3: Detailed Design

DATA FLOW MODEL

The *CDS CD-1.1* is a suite of software components that provides reliable delivery and parsing of CD-1.1 protocol frames. The software is designed to continue execution indefinitely without operator intervention after it is configured and operating. Data are handled in a near-real-time manner by *CDS CD-1.1* components to provide a flow of time-series data to forwarding destinations and signal processing and analysis components of the Monitoring System. The *CDS CD-1.1* software has two operational modes, traditional unicast and multicast.

Traditional Unicast Operation

[Figure 14](#) shows the *CDS CD-1.1* data flow for traditional unicast operation at the IDC. *Connection Manager* and *Connection Manager Server* establish inbound connections to the *CDS CD-1.1* at the IDC with a cooperating system participating in the CD-1.1 protocol. Inbound processing activity of the *CDS CD-1.1* is initiated by a data provider delivering a Connection Request Frame to a “well known host” and port number. This “tickle” of an IP port is handled by the Internet daemon process *inetd* of the UNIX operating system. The Internet daemon responds to the port connection request by starting a *Connection Manager* process and passing the connection request packet received to that process. After validating the request, *Connection Manager* selects a host to service the connection being requested and sends a message to the host requesting the service of *Connection Manager Server*. The communication to and initiation of *Connection Manager Server* is accomplished in the same manner as that used for *Connection Manager* itself; that is, the Internet daemon of the operating system is used to start a process when the known address is tickled. Assuming that a connection is accepted, *Connection Manager Server* responds to *Connection Manager* with the port number to use. The port number is

provided to the data provider in the form of a CD-1.1 Connection Response Frame. After the response frame is sent, *Connection Manager* has fulfilled its objective and exits. It is reactivated by *inetd* when the next Connection Response Frame is received. The activated *Connection Manager Server* waits to be contacted on the port with an Option Request Frame from the data provider. When this frame is received the *Connection Manager Server* responds with an Option Response Frame and *exec*'s an *Exchange Controller* to further service the connection.

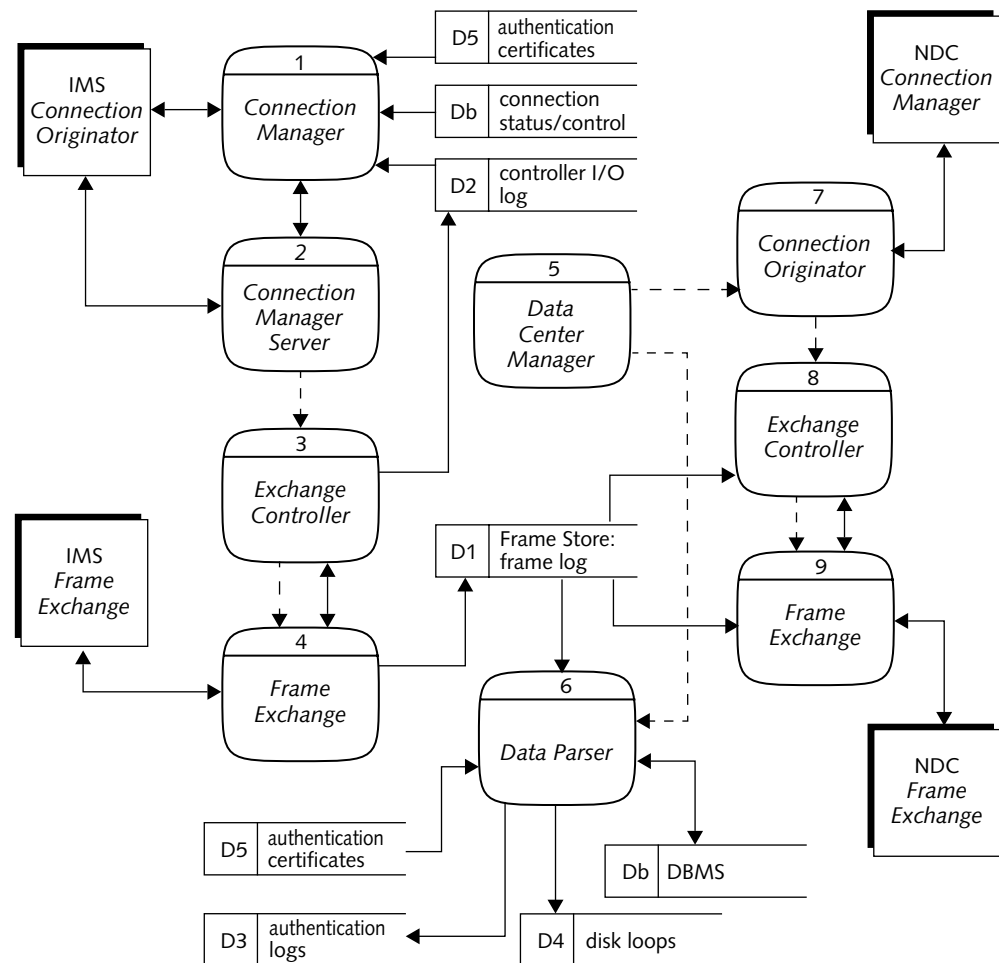


FIGURE 14. DATA FLOW MODEL OF CDS CD-1.1 TRADITIONAL UNICAST OPERATION

▼ Detailed Design

The method used to start *Exchange Controller* is such that *Exchange Controller* assumes/takes-over the process ID of *Connection Manager Server*. The result is that *Connection Manager Server* ceases to exist (essentially the same as an exit) and *Exchange Controller* executes in its place. *Exchange Controller* works in concert with *Frame Exchange* to service a connection to a protocol peer participating in the CD-1.1 protocol. *Exchange Controller's* parent provides a communication port at startup. *Exchange Controller* passes the port to the *Frame Exchange* process that it execs. The forked *Frame Exchange* opens the communication port and attempts to communicate with the protocol peer. When *Frame Exchange* receives the frames, it stores the frames in the Frame Store, notifies *Exchange Controller* about the newly received frames, and updates its accounting for the next AckNack Frame. AckNack Frames are periodically sent to the data provider by *Frame Exchange* to declare its presence and state. In this way the AckNack not only serves as a “heartbeat” indicator, but also acknowledges the frames that have been successfully received. *Frame Exchange* and *Exchange Controller* remain active until a terminating command or condition is encountered. In the event of such a condition, an attempt is made to tell the participating node of the exit condition via an Alert Frame prior to exiting.

Data Center Manager manages the processes responsible for IDC outbound CDS CD-1.1 connections and processing local to the IDC. There is a *Data Center Manager* for each host providing such services to the CDS CD-1.1. *Data Center Manager* is initiated by user input and takes a file that provides configuration information as a command-line argument. After it is activated *Data Center Manager* runs indefinitely or until a terminating condition is encountered, such as the receipt of a kill signal. *Data Center Manager* execs the processes it is to manage (as specified in its configuration file), monitors their progress via connections to the output data streams, and identifies process IDs (PIDs). If a monitored process terminates, *Data Center Manager* has the ability to restart the program (or a different program) according to configuration values.

Data Parser is one of the processes started and managed by *Data Center Manager*. *Data Parser* is responsible for decomposing CD-1.1 Data Frames to extract time-series signal data. *Data Parser* retrieves newly received frames written by *Frame Exchange* from the Frame Store. Channel Subframes are extracted from frames and

verified for authenticity if digital signatures are present. Data contained in the Channel Subframes are converted to the CSS 3.0 data format. Converted data are written to disk loop files, and a DBMS waveform description (**wfdisc**) record is created and stored for the converted data. After they are created, the **wfdisc** record(s) and the disk loop files are used by signal processing and analysis software of the monitoring system. *Data Parser* is designed to execute indefinitely and periodically polls the Frame Store for new frames. More than one *Data Parser* may be executing in the CDS. However, a given data provider must be configured to be processed by only one *Data Parser*.

Data Center Manager initiates and monitors processing to keep data flowing to the configured forwarding destinations. A *Connection Originator* process is started by *Data Center Manager* for each forwarding destination. *Connection Originator* exchanges the Connection Request and the Option Request Frames with a forwarding destination to establish a connection port between the computer systems. The interaction required to establish this connection is the mirror image of the interaction for the inbound connection to the IDC by *Connection Manager* and *Connection Manager Server*. When a connection is granted *Connection Originator* `exec`'s an *Exchange Controller* process and provides the received communication port. When *Exchange Controller* is successfully `exec`'d, *Connection Originator* ceases to exist allowing the child process to assume the process ID and file descriptors of the parent. This processing flow follows the model used by *Connection Manager Server* to establish an inbound connection to the IDC. Because the child process inherits the *Connection Originator* process ID, *Exchange Controller* (and its associated connection) becomes the process monitored by *Data Center Manager*.

Multicast Operation

[Figure 15](#) shows the CDS CD-1.1 data flow for multicast operation at the IDC. The multicast subsystem is shown on the upper part of the figure, and the unicast catchup subsystem is shown on the lower part. The multicast subsystem consists of *Connection Originator* and *Multicast Receiver*. There is no multicast sender at the

▼ Detailed Design

The diagram illustrates the architecture of the NDC system, showing the flow of data and control between various components and databases.

Components and Data Flow:

- IMS Path (Left):**
 - 1 Connection Originator:** Receives data from the **IMS Connection Manager** and the **Data Center Manager** (7). It sends data to the **2 Multicast Receiver**.
 - 2 Multicast Receiver:** Receives data from the **IMS Multicast Sender** and the **1 Connection Originator**. It sends data to the **3 Missing Frame Detector**.
 - 3 Missing Frame Detector:** Receives data from the **2 Multicast Receiver** and the **Data Center Manager** (7). It sends data to the **4 Connection Originator** and the **D4 missing sequence numbers** database.
 - 4 Connection Originator:** Receives data from the **3 Missing Frame Detector** and the **Data Center Manager** (7). It sends data to the **5 Exchange Controller** and the **D4 missing sequence numbers** database.
 - 5 Exchange Controller:** Receives data from the **4 Connection Originator** and the **D4 missing sequence numbers** database. It sends data to the **6 Frame Exchange** and the **D1 Frame Store: frame log** database.
 - 6 Frame Exchange:** Receives data from the **5 Exchange Controller** and the **D1 Frame Store: frame log** database. It sends data to the **IMS Frame Exchange** and the **D1 Frame Store: frame log** database.
- NDC Path (Right):**
 - 7 Data Center Manager:** Receives data from the **1 Connection Originator** and the **3 Missing Frame Detector**. It sends data to the **8 Connection Manager** and the **9 Connection Manager Server**.
 - 8 Connection Manager:** Receives data from the **7 Data Center Manager** and the **D3 authentication certificates** database. It sends data to the **9 Connection Manager Server** and the **D2 controller I/O log** database.
 - 9 Connection Manager Server:** Receives data from the **8 Connection Manager** and the **D2 controller I/O log** database. It sends data to the **10 Exchange Controller** and the **D5 missing sequence numbers** database.
 - 10 Exchange Controller:** Receives data from the **9 Connection Manager Server** and the **D5 missing sequence numbers** database. It sends data to the **11 Frame Exchange** and the **D2 controller I/O log** database.
 - 11 Frame Exchange:** Receives data from the **10 Exchange Controller** and the **D2 controller I/O log** database. It sends data to the **NDC Frame Exchange** and the **D2 controller I/O log** database.

Databases and External Managers:

- D1 Frame Store: frame log**: Receives data from the **5 Exchange Controller** and the **6 Frame Exchange**. It sends data to the **6 Frame Exchange** and the **11 Frame Exchange**.
- D2 controller I/O log**: Receives data from the **8 Connection Manager** and the **10 Exchange Controller**. It sends data to the **9 Connection Manager Server** and the **11 Frame Exchange**.
- D3 authentication certificates**: Receives data from the **8 Connection Manager**. It sends data to the **8 Connection Manager**.
- D4 missing sequence numbers**: Receives data from the **3 Missing Frame Detector** and the **4 Connection Originator**. It sends data to the **3 Missing Frame Detector** and the **4 Connection Originator**.
- D5 missing sequence numbers**: Receives data from the **9 Connection Manager Server**. It sends data to the **10 Exchange Controller**.

External Managers:

- IMS Connection Manager**: Connected to the **1 Connection Originator**.
- IMS Multicast Sender**: Connected to the **2 Multicast Receiver**.
- IMS Frame Exchange**: Connected to the **6 Frame Exchange**.
- NDC Connection Originator**: Connected to the **8 Connection Manager**.
- NDC Frame Exchange**: Connected to the **11 Frame Exchange**.

FIGURE 15. DATA FLOW MODEL OF CDS CD-1.1 MULTICAST OPERATION

Both *Multicast Receiver* and *Frame Exchange* write to the Frame Store. *Data Parser*, which is responsible for decomposing the Frame Store into timeseries data, is not shown. However, it functions in the same manner as in traditional unicast operation.

Multicast Subsystem

The multicast subsystem is responsible for requesting a connection to an IMS station, joining a multicast group, receiving multicast data packets, reconstructing protocol frames, writing frames to the Frame Store and requesting, and receiving missing multicast data packets. If *Connection Originator* or one of its child processes is not running, *Data Center Manager* starts *Connection Originator*. *Connection Originator* sends a Connection Request Frame to *Connection Manager* at the IMS station. If validated, *Connection Manager* responds with a Connection Response Frame containing the multicast group and PNack receiver connection information. *Connection Originator* exec's a *Multicast Receiver* process and provides it with the connection information. When *Multicast Receiver* is successfully exec'd, *Connection Originator* exits, allowing the child process to assume the process ID and file descriptors of the parent. *Multicast Receiver* starts listening for data packets multicast from *Multicast Sender* at the IMS station and stores the packets in an internal circular buffer. When all packets for a protocol frame have been received, *Multicast Receiver* reconstructs the frame and writes it to the Frame Store. If *Multicast Receiver* detects any missing packets, it sends a PNack packet to *Multicast Sender* requesting the missing packets be remulticast. When the circular buffer fills up, it writes over the oldest packets received. If all the packets for a frame are not received before the circular buffer over writes the slots, the frame is abandoned.

Unicast Catchup Subsystem

The unicast catchup subsystem is responsible for requesting and retrieving protocol frames missed by the multicast subsystem. It is also responsible for providing missing frames to other data consumers. The unicast catchup subsystem uses the same components as those used for traditional unicast operation with two major differ-

▼ Detailed Design

ences. In the unicast catchup operation, connections are initiated by the data consumer rather than the data provider. Second, *Frame Exchange* and *Exchange Controller* operate only until the missing frames are retrieved. After the missing frames have been received, *Frame Exchange* and *Exchange Controller* exit. Missing frame sequence numbers are detected by *Missing Frame Detector*.

Data Center Manager periodically starts *Missing Frame Detector*, if it is not currently running. *Missing Frame Detector* reads the Frame Store and looks for missing frames from the sequence. If missing frames are detected, their sequence numbers are saved to a temporary store. *Data Center Manager* monitors the processing result of *Missing Frame Detector* and invokes the *Connection Originator* when missing frames have been detected. *Connection Originator* requests a unicast catchup connection from *Connection Manager* at the IMS station by sending a Connection Request Frame. If *Connection Manager* at the IMS station validates the request, it responds with a Connection Response Frame. *Connection Originator* then sends an Option Request Frame containing a list of missing sequence numbers to be sent. *Connection Manager Server* at the IMS responds with an Option Response Frame. When the Option Response Frame is received, *Connection Originator* execs an *Exchange Controller* process, which in turn execs a *Frame Exchange* process. *Exchange Controller* works in concert with *Frame Exchange* to service the connection as in traditional unicast operation. As frames are received, *Frame Exchange* writes them to the Frame Store and sends acknowledgements to *Frame Exchange* at the IMS. After *Frame Exchange* at the IMS has sent all the missing frames and received acknowledgements, it sends an Alert Frame causing the protocol peers to exit.

In the role of a reliability host, the unicast catchup subsystem may service requests for missing frames from other data consumers. *Connection Originator* at a NDC sends a Connection Request Frame to a well known port number. The Internet daemon process *inetd* responds to the port connection request by starting a *Connection Manager* process and passing the connection request packet received to that process. As in traditional unicast operation, *Connection Manager* validates the request and starts a *Connection Manager Server* process via *inetd*. *Connection Manager Server* responds to *Connection Manager* with the port number to use. *Connection Manager* sends a Connection Response Frame to the data consumer

containing the port number and then exits. *Connection Manager Server* waits to be contacted on the port with an Option Request Frame from the data consumer. The Option Request Frame contains a list of missing sequence numbers to be sent. When this frame is received the *Connection Manager Server* sends an Option Response Frame to *Connection Originator* at the NDC, writes the missing sequence numbers to a file and execs an *Exchange Controller* process. *Exchange Controller* works in concert with *Frame Exchange* to service the connection. *Exchange Controller* reads the missing sequence numbers from the file and starts *Frame Exchange*. *Frame Exchange* reads the missing frames from the Frame Store and sends them to *Frame Exchange* at the NDC. After *Frame Exchange* at the NDC has received and acknowledged all the requested frames, *Frame Exchange* sends an Alert Frame causing the protocol peers to exit.

PROCESSING UNITS

The CDS CD-1.1 consists of the following processing units:

- *Connection Manager*
- *Connection Manager Server*
- *Data Center Manager*
- *Connection Originator*
- *Exchange Controller*
- *Frame Exchange*
- *Multicast Sender*
- *Multicast Receiver*
- *Missing Frame Detector*
- *Data Parser*
- *Frame Store Stager*
- *Protocol Checker*
- *libfs*
- *libcdo*

▼ Detailed Design

The following paragraphs describe the design of these units, including any constraints or unusual features in the design.

Connection Manager

Connection Manager negotiates a network connection with a requesting protocol peer. *Connection Manager* reads a Connection Request Frame from the requester, validates the request, and sends back a Connection Response Frame with connection information. [Figure 16](#) shows the *Connection Manager* context.

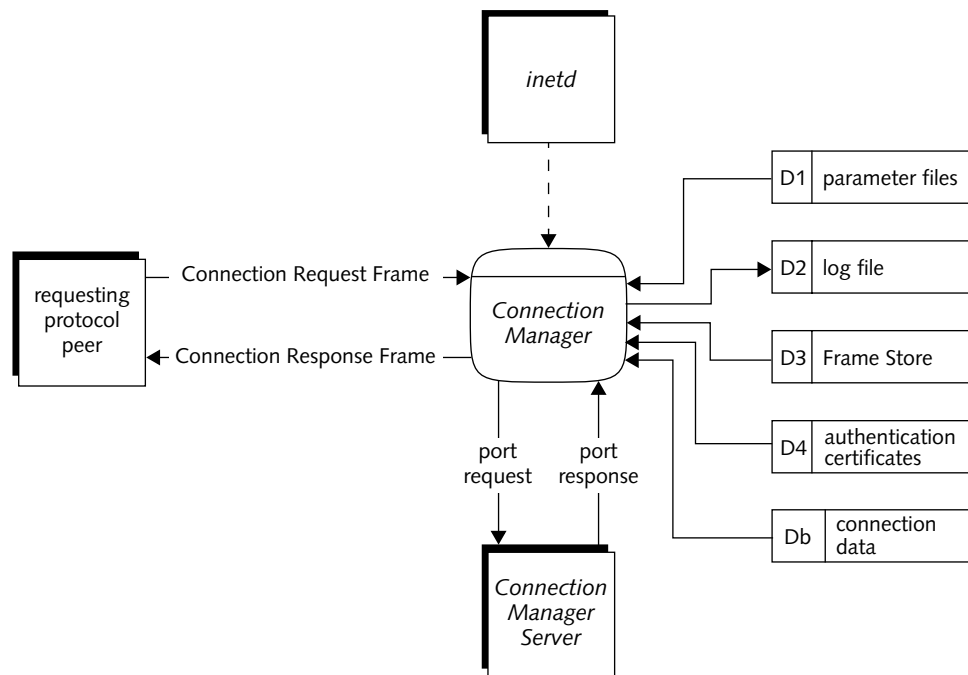


FIGURE 16. CONNECTION MANAGER CONTEXT

Input/Processing/Output

The following are inputs to *Connection Manager*:

- Run-time configuration information. This information may be provided on the command line, but are most conveniently contained in a parameter file. Parameters define operational behavior of *Connection Manager* including time-out values, names of log files, and database accounts. Configuration parameters are accessed with routines of the *libpar* library.
- Connection Request Frame. This frame is received via socket communications from a protocol peer.
- Recognized protocol peer. This information may be stored in database tables and, if so, is accessed with *libgdi* library functions when validating connection requests. In the absence of a DBMS, data are maintained in files.
- Active connection information. This information is identified by checking time tags in Frame Store entries written by the *Exchange Controller*. These entries are accessed with routines of the *libfs* library.
- Port response message. This message is received via socket communications from *Connection Manager Server* and contains a socket port number for further data transmission.

Processing

Processing of *Connection Manager* is provided in the following components:

- *Connection Manager Processing*
- *Frame Processing*
- *Database Processing*
- *Socket Processing*

[Figure 17](#) shows the interaction of these components.

▼ Detailed Design

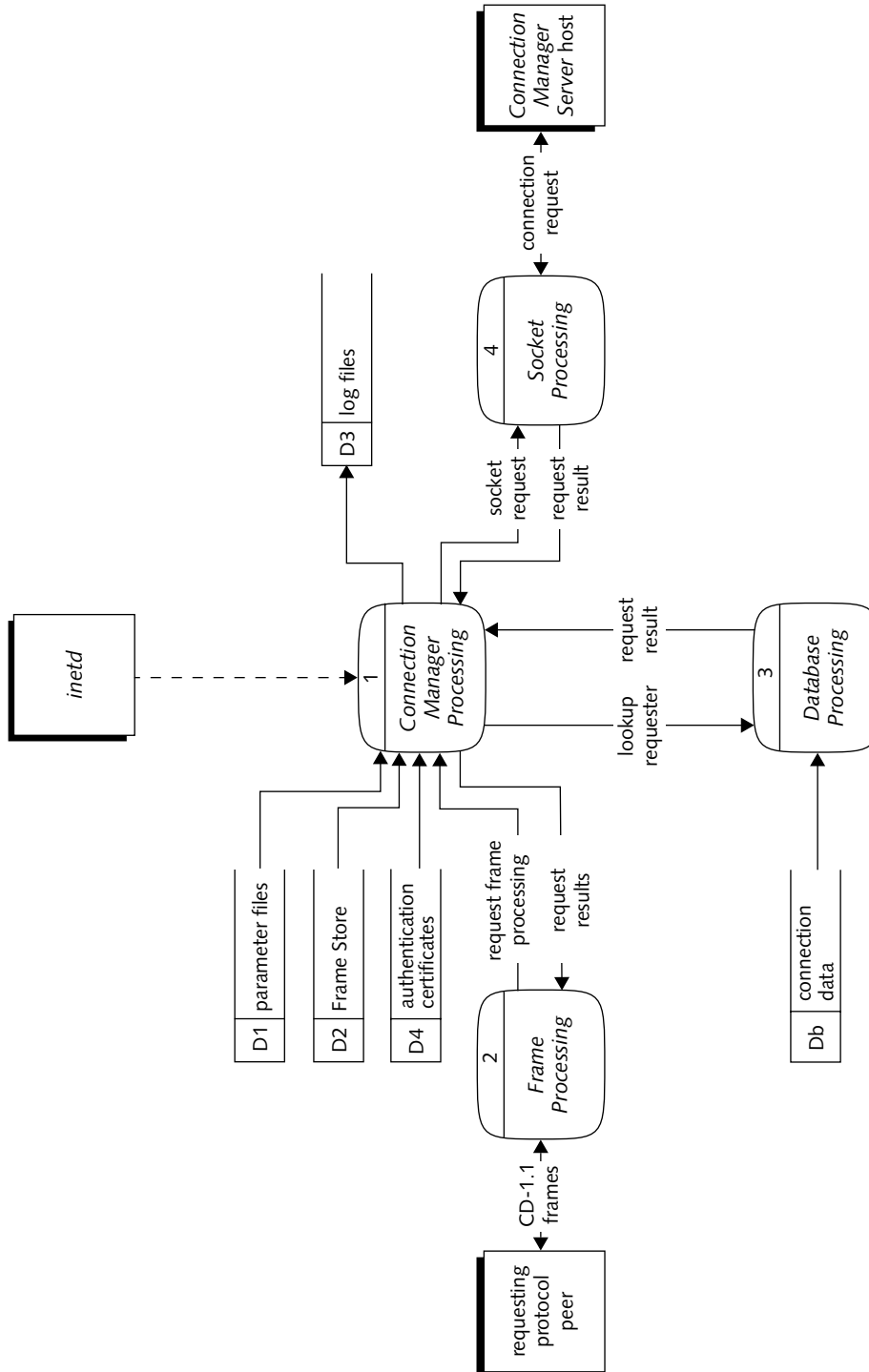


FIGURE 17. CONNECTION MANAGER COMPONENTS

Connection Manager Processing

Connection Manager Processing is the component that coordinates activities of the other *Connection Manager* components. *Connection Manager Processing* initializes the logging capability for status messages and error messages. It also initializes configuration parameter inputs, signal handling, and authentication attributes.

If *inetd* is being used, the data are read from standard input. If *inetd* is not being used, processing in *Socket Processing* is used to open and read data from a TCP/IP socket. *Frame Processing* is used to determine if the data read is a Connection Request Frame, verify that the signature of the frame is valid, and unpack the frame data. If the database is being used, *Database Processing* is invoked to verify the requester. Processing then verifies that the requester/provider is not already connected by looking in the Frame Store.

For traditional unicast and unicast catchup connections, a connection server must be selected. *Connection Manager* obtains a list of *Connection Manager Server* hosts by using either *Database Processing* or configuration parameters, and then loops through each listed host. *Socket Processing* is used to attempt a socket connection. If a connection cannot be made, the software tries the next host in the list. If a connection is made, a port request message is sent to the responding *Connection Manager Server* and processing waits for a port response message. If no response is received within the configured time-out, the software tries the next host in the list. When a valid port response is received, the port is verified by a connection attempt using *Socket Processing*. After the selection of a *Connection Manager Server* host, *Frame Processing* is used to build and send a Connection Response Frame containing the server's host address and port number to the requester.

For multicast connections, *Connection Manager* runs on a station data provider. Connection information is retrieved from configuration data. *Frame Processing* is then used to build and send a Connection Response Frame containing the connection information to the requester.

Frame Processing

Frame Processing provides a group of service that handle frame I/O, frame verification, and frame construction. Frame I/O provides the ability to read and write frames from/to a socket and log a message if there is an error.

Connection Request Frames are checked by:

- verifying the frame type
- checking the authentication signature
- unpacking the frame
- checking the station type
- checking the service type

For traditional unicast and unicast catchup connections, Option Request Frames are checked by:

- verifying the frame type
- checking the authentication signature
- unpacking the frame
- checking the body count
- checking the option type
- checking the requester's name

Frame construction capability is provided for Connection Response Frames and Option Response Frames using functions in *libcdo*.

Database Processing

Database Processing provides a group of low-level service routines. These routines handle the required interaction between the *Connection Manager* and the DBMS through the *libgdi* API. The following capabilities are provided:

- Open and close database connections.
- Obtain a list of server hosts from the **dlman** table, and order the list according to preferred host.

- Provide locking capability on a data provider's entry in the **alphasite** table; this ability is used to ensure that multiple connection requests are not processed for the same requester.
- Look-up a connection requester's name and address in the **alphasite** table.

Socket Processing

Socket Processing provides a group of low-level service routines that provide access to the socket I/O by requesting components. The following capabilities are provided:

- Establish inbound connection, that is, listen to a port and accept connections.
- Establish outbound connections.
- Examine the address of a named host.
- Provide the IP address of a local host.
- Read and send port request messages.

Output

Outputs of *Connection Manager* are port messages, CD-1.1 Connection Response Frames, database updates, and log messages:

- Port messages are sent via sockets to the *Connection Manager Server* host to request a connection port.
- Connection Response Frames are returned to requesting protocol peers when a connection is accepted. (When a connection request is denied, no response is provided to the requester.)
- Errors and significant processing events are written to *Connection Manager's* log file to record processing activity.

▼ Detailed Design

Control

Connection Manager is started by *inetd* when a Connection Request Frame is received at the well-known host on the well-known port and terminates after responding to the request. In the case of a successful connection a Connection Response Frame is provided to the requester. In the case of an unsuccessful connection no response is provided.

Interfaces

- *Connection Manager* interfaces with connection requesters using Connection Request Frames across a TCP/IP socket connection.
- *Connection Manager* interfaces with *Connection Manager Server* using a TCP/IP socket connection. Port request and port response messages are exchanged over this interface to coordinate connection host and port number.
- *Connection Manager* interfaces with the DBMS using *libgdi* library functions.
- *Connection Manager* interfaces with the host file system using *liblog* functions to log error conditions and significant processing events.

Error States

Execution of *Connection Manager* concludes with either a successful or unsuccessful connection request. An unsuccessful attempt is attributed to the following possibilities:

- failed execution
- successful execution that resulted in a denied request

Failed execution is most generally attached to configuration errors or an interface failure. Examples of configuration errors include:

- incorrect database account name
- protocol peers not defined in the **alphasite** table

- missing public key for authentication

Examples of interface failures include:

- ports to *Connection Manager Server* not available
- *Connection Manager Server* process not responding

Unsuccessful connection requests are expected when the request does not adhere to CD-1.1 protocol. Examples of these conditions include:

- unrecognized protocol peer requests a connection
- connection request received from a protocol peer that is already connected
- connection request received from a legitimate protocol peer that recently terminated a connection, and the configured time-out period between connections has not yet expired

Connection Manager Server

Connection Manager Server works with *Connection Manager* to establish traditional unicast or unicast catchup connections for a requesting protocol peer. *Connection Manager Server* is not used for multicast connections. *Connection Manager Server* processes port requests from *Connection Manager*, exchanges Option Request/Response Frames with the connection requester, and starts *Exchange Controller* on the server. [Figure 18](#) shows the *Connection Manager Server* context.

▼ Detailed Design

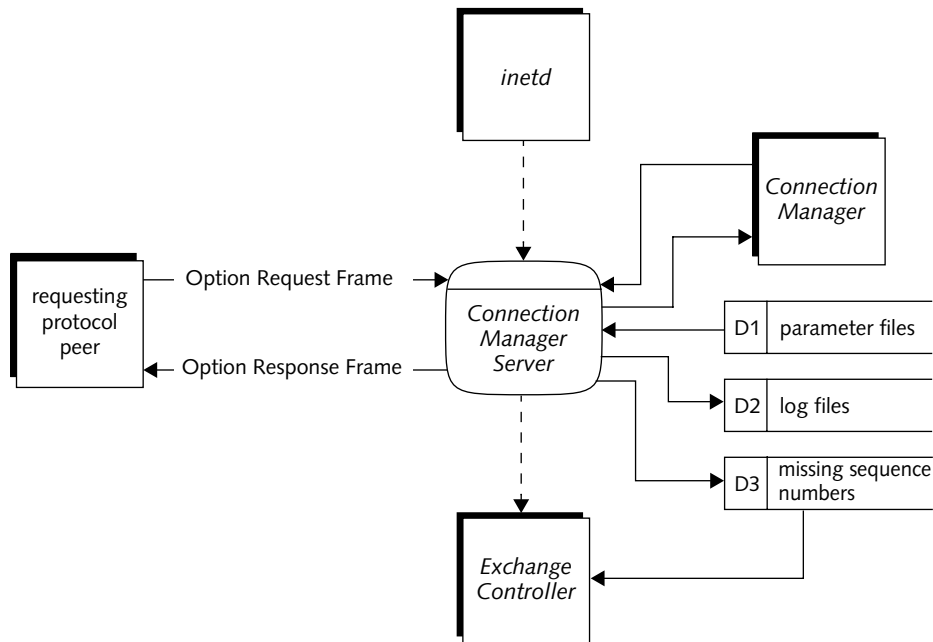


FIGURE 18. CONNECTION MANAGER SERVER CONTEXT

Input/Processing/Output

The inputs to *Connection Manager Server* are as follows:

- Run-time configuration information provides configuration data in the parameter files. Parameter files provide information to describe logging and operational behavior, and to identify valid connection ports for the server host. Configuration parameters may be specified on the command line but are provided most conveniently in the files themselves. Methods of the *libpar* library are used to access configuration parameters.
- Port request messages are sent from *Connection Manager* via socket communications to *Connection Manager Server*. The message is a request for the identifier of a socket that may be used for future data transmissions to the host by a requesting protocol peer.

- Option Request Frames are sent from a protocol peer requesting a connection. This CD-1.1 frame is received via socket communication to finalize the connection negotiation between the local and remote protocol peers. In catchup operation, Option Request Frames contain a list of missing sequence numbers to be sent.

Connection Manager Server processing provides the following functions:

- initializing program operations
- responding to a port request
- exchanging Option Request/Response Frames with a requesting protocol peer
- in unicast catchup operation, extracting the missing sequence numbers from the Option Request Frame and writing them to a UNIX file
- starting *Exchange Controller*

Connection Manager Server processing shares components (code) with *Connection Manager*. Shared components are *Frame Processing* and *Socket Processing*. Descriptions of these components are provided in the section on the [“Connection Manager” on page 54](#). [Figure 19](#) shows the interaction of the *Connection Manager Server* components. Initialization and configuration processing initializes logging capability for status and error messages. Processing is also provided to read configuration parameter input, initialize signal handling, and read authentication attributes.

Connection Manager Server responds to a port request message from *Connection Manager*. If *inetd* is being used, data are read from standard input. If *inetd* is not used, processing in *Socket Processing* is used to open and read the data from a TCP/IP socket. The received port request message provides the name of the requesting protocol peer and requests a TCP/IP connection port for further data transmission. *Socket Processing* looks up the requester’s name in a list of peers that this process instance is configured to support. If the requester is recognized, that is, configured to be supported by this *Connection Manager Server*, a port response message is created and is sent back to *Connection Manager* via standard out (`stdout`) or a socket, depending on how the request message was received. If the

▼ Detailed Design

host is not configured for the requester, the port response message is not created or sent. The port response message provides the port number that the requester uses for further communication with *Connection Manager Server* on the executing host. *Connection Manager Server* does not attempt to validate that multiple port requests for the same protocol peer are not granted. *Socket Processing* depends on *Connection Manager* to shield it from such occurrences.

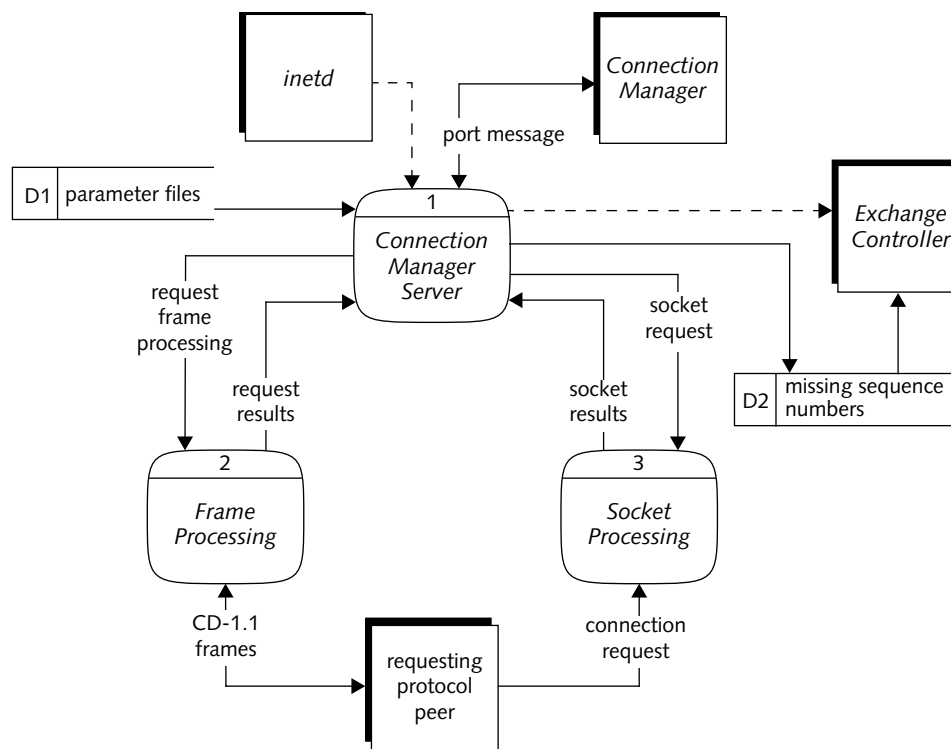


FIGURE 19. CONNECTION MANAGER SERVER COMPONENTS

Connection Manager Server negotiates protocol options between protocol peers. Option Request/Response Frame processing proceeds as follows:

1. *Connection Manager Server* uses *Socket Processing* capabilities to listen for connections to the port provided in the port response message.

2. If no connection is made within a configured time-out interval, *Connection Manager Server* terminates.
3. When a connection is made *Frame Processing* capabilities are used to read, verify, and identify an Option Request Frame.
4. The requester's name provided in the Option Request Frame is verified against the name received in the port request message.
5. The option specified is examined to verify compliance with the local site's capabilities.
6. The frame processing constructs and sends an Option Response Frame to the requesting protocol peer over the now connected socket.
7. If the connection was correctly established, the open socket is the port used for further transmissions between the local and remote protocol peers. If not, the program logs the error and exits.
8. For unicast catchup connections, *Connection Manager Server* reads the list of missing frame sequences from the Option Request Frame and writes them to a file.
9. *Connection Manager Server* starts the *Exchange Controller* program and passes it the open socket file descriptor as a command-line argument.

Connection Manager Server outputs are port response messages, CD-1.1 Option Response Frames, missing frame sequence numbers, and log messages:

- Port response messages are sent via socket communications to *Connection Manager* to provide an available connection port identifier.
- Option Response Frames are returned to requesting protocol peers in response to received Option Request Frames.
- For unicast catchup connections, missing frame sequence numbers are written to a file for input to *Exchange Controller*.
- Errors and significant processing events are written to *Connection Manager Server's* log file using *liblog* functions to record processing activity.

▼ Detailed Design

Control

- *Connection Manager Server* is started by *inetd* when a *Connection Manager* sends a port request message to a well-known host on the well-known port.
- If a successful connection is established, *Connection Manager Server* terminates after starting the *Exchange Controller* process. When *Exchange Controller* is started it overlays *Connection Manager Server* and assumes its process ID (PID).
- If a connection is not established *Connection Manager Server* logs the terminating condition to its log file and exits.

Interfaces

- *Connection Manager Server* interfaces with *Connection Manager* using a TCP/IP connection. Data over this interface are port request and port response messages.
- The interface to a requesting protocol peer is over a TCP/IP socket connection. This interface is used for CD-1.1 Option Request and Option Response Frames.
- The host file system interface is managed by *liblog* functions and is used to log error conditions and significant processing events.
- *Connection Manager Server* has a control interface with *Exchange Controller* and uses the system service *exec* to start the process.

Error States

Connection Manager Server execution results in either an established or undetermined connection to a protocol peer. Unsuccessful connection attempts are most likely the result of configuration errors. Probable configuration errors include:

- The peer host name is not identified in the port configuration par file.
- The public key for authentication is missing.
- The path to the *Exchange Controller* executable is missing or incorrect.

- The path to the *Exchange Controller* parameter file is missing or incorrect.

A request is also denied if an Option Request Frame is received from an unanticipated source or for an unsupported option. This is not an error condition, but a legitimate validation failure.

Data Center Manager

Data Center Manager keeps other CDS CD-1.1 components running. It starts, monitors, and restarts processes on host computers. [Figure 20](#) shows the context of *Data Center Manager*. Managed processes at the data center include *Data Parser*, *Connection Originator*, and *Missing Frame Detector*. Because *Exchange Controller* and *Frame Exchange* run as descendants of *Connection Originator*, *Data Center Manager* shares partial responsibility for keeping these processes in operation. *Data Parser*, *Connection Originator*, and *Missing Frame Detector* are categorized as child processes in the context diagram ([Figure 20](#)). There are currently no processes configured as command and control processes in [Figure 20](#).

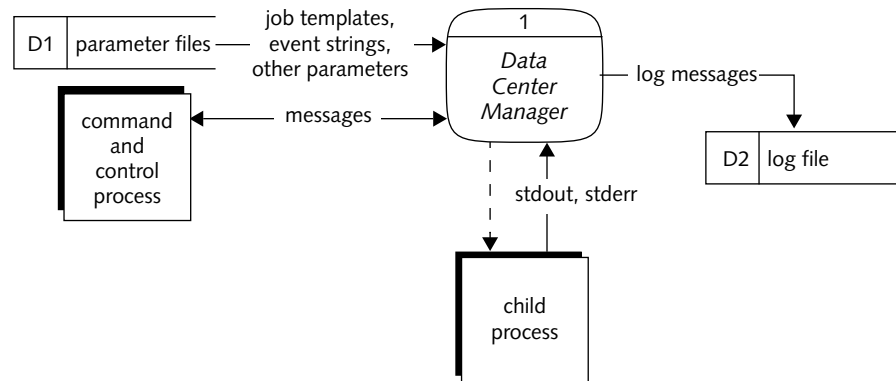


FIGURE 20. DATA CENTER MANAGER CONTEXT

Data Center Manager's processing is centered around a queue of events, and its internal structure reflects this model. An event for *Data Center Manager* is a processing condition that requires action. Events may be synchronous or asynchro-

▼ Detailed Design

nous. Three of *Data Center Manager's* internal objects wait for asynchronous system events, translate them into *Data Center Manager* events, and enqueue the events in an *Eventqueue* object (see [Figure 21](#)).

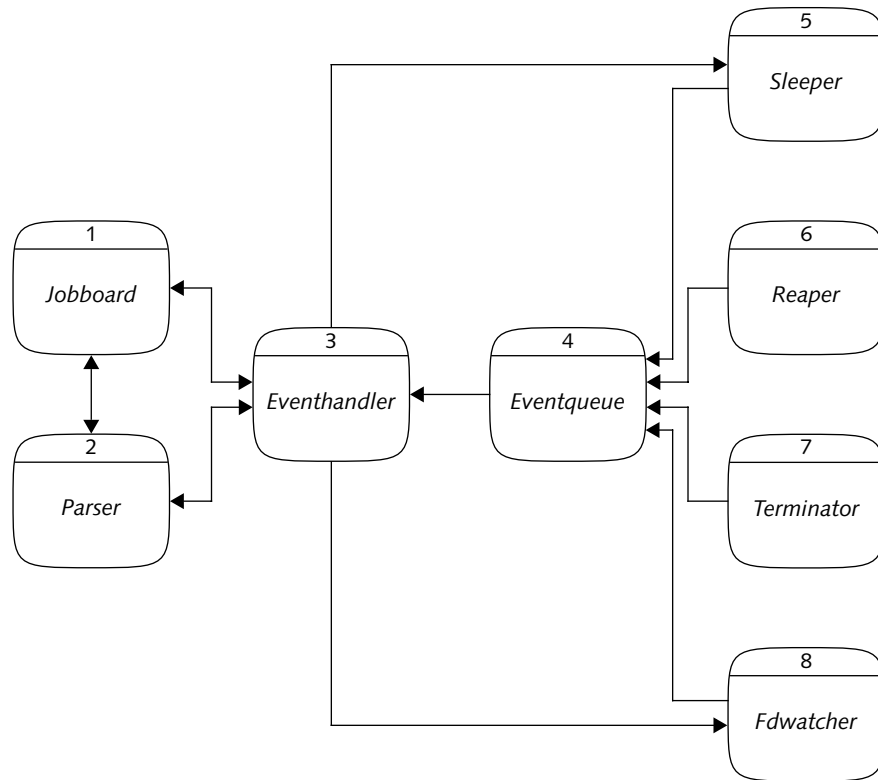


FIGURE 21. DATA CENTER MANAGER PROCESSING COMPONENTS

- The *Terminator* object translates SIGTERM and SIGINT signals sent to *Data Center Manager* into HaltRequest events.
- The *Reaper* object waits for *Data Center Manager's* child processes to exit, and when they do, creates and enqueues ChildExited events.

- The *Fdwatcher* object waits for file descriptors to become active, then creates an *FDAActive* event for each active descriptor detected. File descriptors are not monitored before they are given to *Fdwatcher* by *Eventhandler* or after their *FDAActive* events are enqueued.

Data Center Manager object *Sleeper* also puts events in *Eventqueue*. Rather than creating events itself, *Sleeper* receives events from *Eventhandler* with a time value indicating how long *Sleeper* is to wait before enqueueing.

The *Eventhandler* object removes events from *Eventqueue* and processes them. Events are jobs that must be performed. To carry out the required processing, *Eventhandler* retrieves and updates information about the job on *Jobboard*. Job templates and the current state of *Data Center Manager*'s managed jobs are kept in the *Jobboard* object.

Other sources of events for *Eventhandler* are *par* file definitions and messages from command and control processes. The *Data Parser* object is used in both of the following situations:

- when *Jobboard* is initialized and converts
`fm-<jtid>-command-on-failure` parameters into events
- when input from child processes capable of providing control messages are captured/read.

Input/Processing/Output

Data Center Manager's inputs include messages received from command and control processes, standard out and standard error (`stderr`) and file output produced by its managed processes, and configuration parameters defined through the standard *libpar* interface. Its outputs include error and status logging as well as messages sent to command and control processes ([Figure 20 on page 67](#)).

▼ Detailed Design

Configuration parameters for *Data Center Manager* are read at startup time. One special set of parameters defines *Data Center Manager*'s "job templates." A job template is a tuple consisting of the name of an executable program, some of its command-line arguments, and other attributes that describe how *Data Center Manager* should run the program (see [Table 2](#)).

A "job" is one instantiation of a job template. For each job, *Data Center Manager* execs a new process that executes the job's program. *Data Center Manager* waits for the new process to exit and then repeats the fork/wait sequence. *Data Center Manager* may concurrently manage multiple jobs instantiated from the same or different job templates.

The `fm-initial-events` parameter provides *Data Center Manager* with a list of initial event strings. Event strings are the textual representations of "Events Strings" processed by *Data Center Manager* (see [Table 3](#)).

In addition to configuration parameter inputs, *Data Center Manager* accepts TCP/IP connections from "command and control" processes on a well-known port of *Data Center Manager*'s host machine. Through these connections, the command and control processes send messages containing event strings to *Data Center Manager*, and *Data Center Manager* sends status messages to the command and control processes.

TABLE 2: DATA CENTER MANAGER JOB TEMPLATE ATTRIBUTES

| Configuration Parameter Name | Attribute Description |
|--|--|
| <code>jt看id</code> | Job template ID. A character string that uniquely identifies the job template. It is used to refer to the job template in the configuration parameters and in event strings. |
| <code>fm-<i><jt看id></i>-exec-path</code> | Name of the program to be run and possibly some initial command-line arguments. |
| <code>fm-<i><jt看id></i>-min-runtime</code> | Minimum amount of time the program must run before it is considered to have exited normally. |

TABLE 2: DATA CENTER MANAGER JOB TEMPLATE ATTRIBUTES

| Configuration Parameter Name | Attribute Description |
|---|---|
| <code>fm-<jtid>-restart-waits</code> | List of times to wait before restarting the program each time it exits abnormally. |
| <code>fm-<jtid>-max-restarts</code> | Maximum number of times a program can constructively exit abnormally. |
| <code>fm-<jtid>-command-on-failure</code> | List of event strings for <i>Data Center Manager</i> to parse and handle when a program has exited abnormally too many times. |

TABLE 3: DATA CENTER MANAGER EVENTS

| Event | Data Center Manager's Action |
|--|---|
| JobRequest | Instantiate a new job and schedule its program to be run. |
| HaltRequest (first) | SIGTERM all running child processes and schedule a second HaltRequest. |
| HaltRequest (next) | SIGKILL any remaining child processes, and exit. |
| StatusRequest | Send a status message to the requesting command and control process. |
| RunJob | Fork a new child process, and have the child execute the job's program. |
| FDAActive (child pipe) | Read from the pipe (the children's <code>stdout</code> / <code>stderr</code>), and log what is read. |
| FDAActive (connection request socket) | Accept the connection, creating a new file descriptor. |
| FDAActive (command and control socket) | Read a message from the socket, parse the event string it contains, and handle the resulting event. |
| ChildExited | Either schedule the job's program to be restarted, or remove the job; when a job is removed the job template may specify a sequence of events for <i>Data Center Manager</i> to handle. |

▼ Detailed Design

Data Center Manager follows an event loop processing model: it maintains a queue of event objects. In each iteration of its loop, it dequeues one event and handles the event based upon the event's type, attributes, and *Data Center Manager*'s current internal state. In the course of handling one event, other events may be created. *Data Center Manager* handles these events before returning to the top of the main loop.

Control

Data Center Manager can be started by a system operator from the command line, or a UNIX process such as *init* can be configured to start *Data Center Manager* at system boot time.

Data Center Manager uses signals for interprocess control. It may be notified of a child process' exit with a SIGCHLD signal. A system operator may halt *Data Center Manager*'s processing by sending it a SIGTERM or SIGINT. As part of *Data Center Manager*'s shutdown procedure, it sends SIGTERM signals to any running child processes, or SIGKILL signals if the processes do not respond to the SIGTERMs.

Data Center Manager terminates when it handles a HaltRequest (see [Table 3](#)). SIGTERM and SIGINT signals sent to *Data Center Manager* are converted internally to HaltRequest. *Data Center Manager* may also receive a message containing a HaltRequest from a command and control process.

Interfaces

External interfaces for *Data Center Manager* are as follows:

- A command line interface is used to start and terminate *Data Center Manager*. This interface may also be exercised/executed from a shell script.
- *Data Center Manager* has a control interface with processes it manages. Startup of processes occur through the fork and execute facilities of the operating system. Termination is affected with process signals.

- *Data Center Manager* uses standard error and standard out data streams as an interface mechanism with managed processes to receive input.
- A TCP/IP socket connection interface is provided to interface with external processes. No processes currently use this facility.
- The host file system interface is managed by *liblog* functions and is used to log error conditions and significant processing events.

Data Center Manager internal interfaces are between execution threads. Five threads of control exist in *Data Center Manager*. Each of the four objects on the right side of [Figure 22](#) (*Sleeper*, *Reaper*, *Terminator*, and *Fdwatcher*) has one thread running in it. Each of these four threads waits for the asynchronous system event for which their object is responsible. When a system event is detected, a corresponding event object is created, and an *Eventqueue* method is used to queue the event.

The fifth thread runs the *Eventhandler* loop. This thread has the following interfaces:

- The *Eventqueue* method is used to dequeue events.
- The *Jobboard* and *Data Parser* methods are used to process events.
- The *Sleeper* method is invoked to schedule an event for processing at a later time.
- The *Fdwatcher* method is used to put file descriptors in *Fdwatcher's* monitored descriptor set.

Contention between threads for shared resources is controlled by the following three mutexes.

- One mutex guards *Eventqueue* and is used by all five threads.
- The *Sleeper* object mutex mediates between the *Eventhandler* thread and the *Sleeper* thread, to control access to the set of events for delayed enqueueing.
- The *Fdwatcher* object controls shared access to the list of watched file descriptors by *Eventhandler* and *Fdwatcher* threads.

▼ Detailed Design

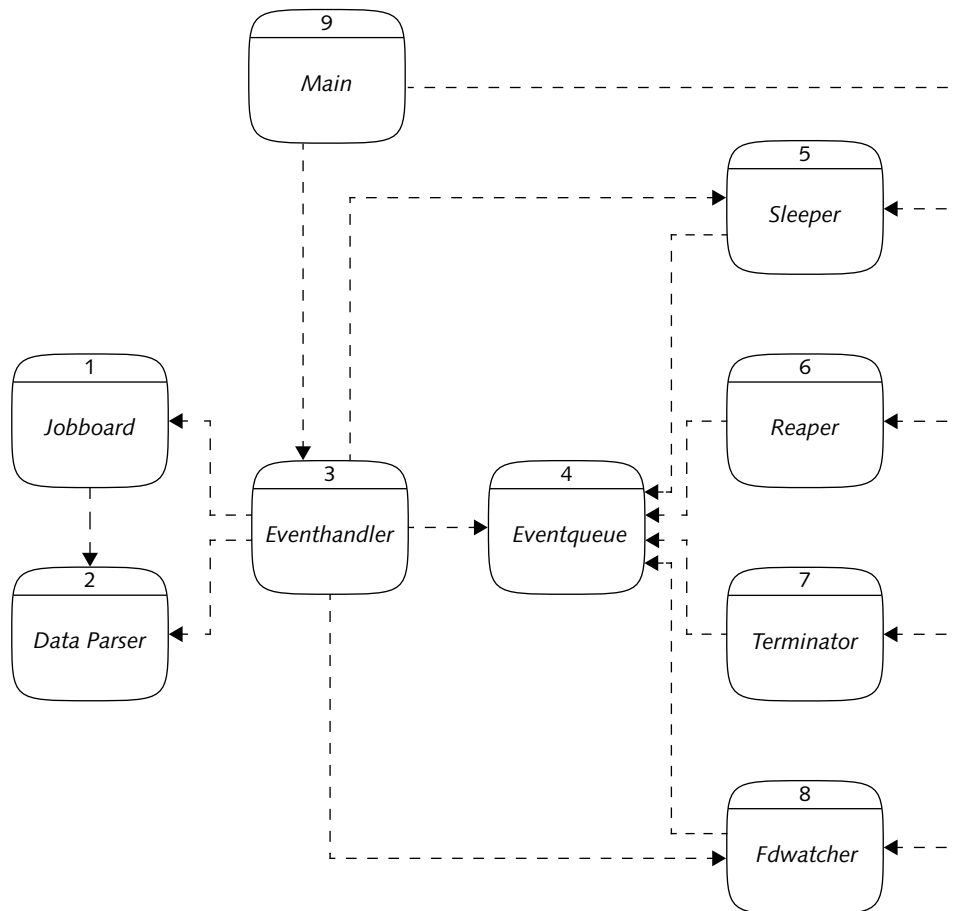


FIGURE 22. DATA CENTER MANAGER INTERNAL CONTROL FLOW

Error States

Data Center Manager is most likely to fail at initialization because of erroneous configuration parameters. The log files usually provide diagnostic information for this type of failure (if *Data Center Manager's* logging component can be initialized before the failure).

A *Data Center Manager* failure may result from exceeding the configured number of controlled jobs, which may be a configuration error. For instance, a job template may specify that two new jobs be instantiated to replace a failed job. Consequently, the number of jobs could quickly grow too large.

Data Center Manager's interface with command and control processes is another potential failure mode. The protocol for the exchange of messages between *Data Center Manager* and the command and control processes is not currently used.

Connection Originator

Connection Originator negotiates a TCP/IP connection using the CD-1.1 protocol ([Figure 23](#)). For traditional unicast and unicast catchup operation, the TCP connection is used for connection, option, and data frames. For multicast operation, it is used only for connection frames. *Connection Originator* then initiates a child process to continue protocol exchanges. Child processes are either *Exchange Controller* or *Multicast Receiver*.

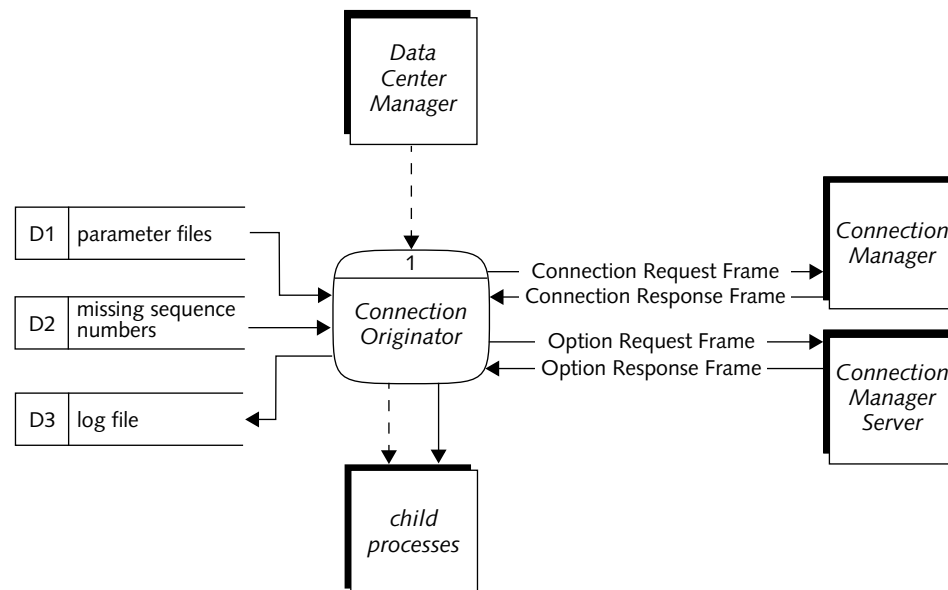


FIGURE 23. CONNECTION ORIGINATOR CONTEXT

▼ Detailed Design

Exchange Controller is used in traditional unicast and unicast catchup modes and inherits the TCP connection established by *Connection Originator*. *Multicast Receiver* is used for multicast operation. It makes no use of the TCP connection. *Connection Originator* consists of the following components:

- *ConnOrig* contains the entry point for *Connection Originator* and is responsible for initialization and overall control of the process.
- *co_portinfo* provides methods for translating configuration parameters into Internet port addresses.
- *co_tcp* contains methods for negotiating a connection using the TCP/IP.
- *co_frames* provides the ability to manipulate and create CD-1.1 frames.
- *co_exhcntl* provides methods for executing child processes.

[Figure 24](#) shows the relationships of these components.

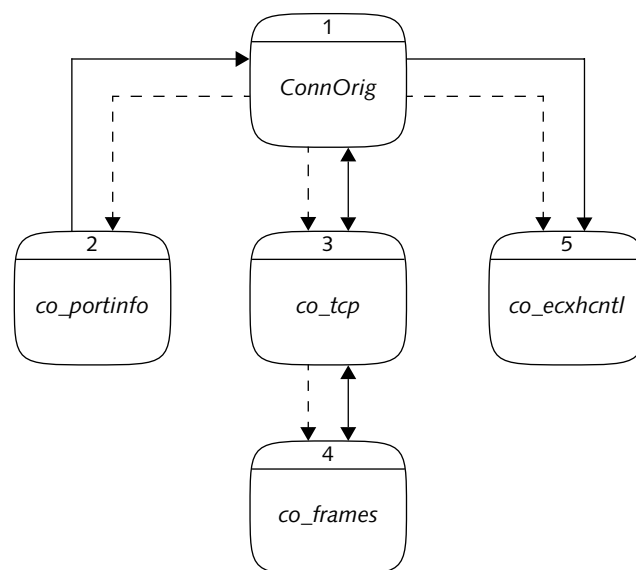


FIGURE 24. CONNECTION ORIGINATOR INTERNAL DATA AND CONTROL FLOW

Input/Processing/Output

Normal processing for *Connection Originator* consists of the following steps:

1. Read the configuration parameters through the *libpar* interface.
2. Establish a preliminary TCP/IP connection with a *Connection Manager* specified in the configuration parameters.
3. Send *Connection Manager* a Connection Request Frame over the preliminary connection.
4. Receive a Connection Response Frame from *Connection Manager* over the preliminary connection.
5. Close the preliminary connection.
6. Establish the main TCP/IP connection with *Connection Manager Server*, whose address is contained in the Connection Response Frame.
7. For unicast catchup connections, read the missing sequence number list and construct an Option Request Frame.
8. Send an Option Request Frame to *Connection Manager Server* over the main connection.
9. Receive an Option Response Frame from *Connection Manager Server* over the main connection.
10. Execute child process, passing control parameters as *libpar* parameters.

Control

Connection Originator is started by *Data Center Manager*. If *Connection Originator* fails to originate its connection, it terminates. If it succeeds, its process becomes the initiated child process using the UNIX `exec` command.

Interfaces

Connection Originator includes the following external interfaces:

- An execution interface with *Data Center Manager* provides startup control.

▼ Detailed Design

- The interface to the connection host (data consumer) is over a TCP/IP socket connection. This interface is used for CD-1.1 Connection Request, Connection Response, Option Request, and Option Response Frames.
- The host file system interface is managed by *liblog* functions and is used to log error conditions and significant processing events.
- *Connection Originator* has a control interface with *Exchange Controller* and *Multicast Receiver* and uses `exec` to instantiate the process.

Internal interfaces of *Connection Originator* are accomplished via C language function calls.

Error States

Connection Originator can fail in the following ways:

- cannot make sense of its configuration parameters
- cannot establish one of its TCP/IP connections
- cannot exchange valid frames with *Connection Manager* or *Connection Manager Server*
- cannot start *Exchange Controller*
- cannot start *Multicast Receiver*

Each of these types of failure is logged in a *Connection Originator* log file. Although failure to read configuration parameters and failure to execute child processes are always fatal, *Connection Originator* can be configured to retry its connection attempts if they fail and to specify fallback addresses to which a connection should be attempted if the first address appears to be unreachable.

Exchange Controller

Exchange Controller provides policy control to the protocol frames handled by *Frame Exchange*. [Figure 25](#) shows the context of *Exchange Controller*. *Frame Exchange* and *Exchange Controller* always exist as a processing pair where the pair is instantiated once for each outbound or inbound connection at the IDC.

Exchange Controller is composed of three components:

- *Controller Executive*
- *Exchange Interface*
- *Frame Handler*

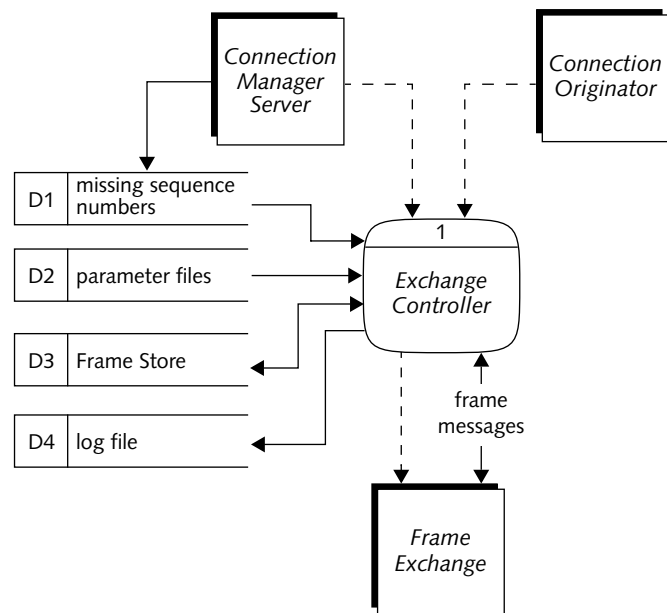


FIGURE 25. EXCHANGE CONTROLLER CONTEXT

[Figure 26](#) shows the data flow between these components. Frame References (frame-ref) and Frame Requests (frame-req) are passed among several components. The following paragraphs present the design of each of these subcomponents.

▼ Detailed Design

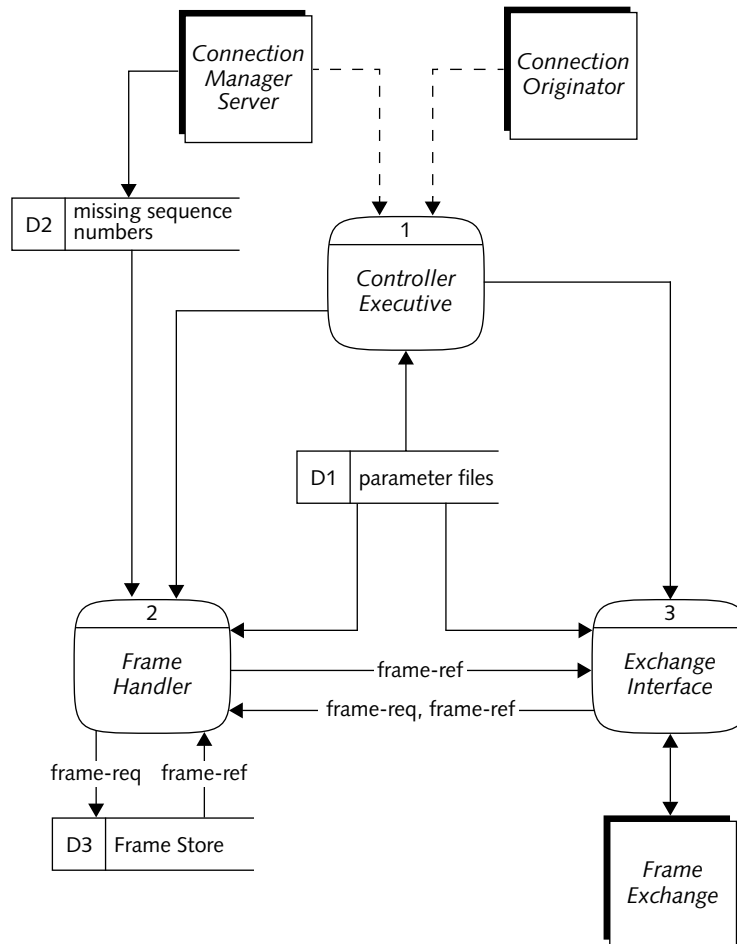


FIGURE 26. EXCHANGE CONTROLLER DATA FLOW

Controller Executive coordinates the processing of *Exchange Controller*. *Controller Executive* is responsible for initializing each of the sibling objects in *Exchange Controller*. After initialization, *Controller Executive* begins a control cycle wherein *Frame Handler* and *Exchange Interface* components are invoked to carry out their processing. When the *Exchange Controller* is terminated, *Controller Executive* orchestrates an orderly cleanup and shutdown of the other processing objects.

Exchange Interface implements the software interface to *Frame Exchange*. This component sends frames and frame messages to *Frame Exchange*, and receives frame messages from *Frame Exchange*. Frame messages are examined for content and acted upon by *Exchange Interface*. In general, frame messages result in calling specific functions to process this class of frame message. The functions provided by *Exchange Interface* are service-level functions with limited knowledge of the content or reason for the data they handle.

Frame Handler manages Data Frames for *Exchange Controller*. *Frame Handler* provides functions for accessing the Frame Store including: opening for access, polling for unacknowledged frames (for startup), and polling for newly arrived frames (in traditional unicast operation). *Frame Handler* is responsible for ordering frames to be sent in a proper order, then supplying the frames to *Exchange Interface* for sending.

Controller Executive

Controller Executive provides the main entry point for *Exchange Controller*. *Exchange Controller* is initiated by either *Connection Originator* or *Connection Manager Server* based on whether an outbound or inbound connection is to be serviced, respectively. The UNIX command-line style command to start the process provides a communication socket file descriptor, connection source/destination indicator, a parameter file designation, and, in the case of unicast catchup, a missing frame sequence numbers file designation. The parameter file supplied to *Exchange Controller* provides run-time configuration values for customizing the execution of the process. The configuration parameter file used by *Exchange Controller* is shared with *Frame Exchange*, which aids in synchronizing the processing of the two processes. At startup *Exchange Controller* reads configuration values from the command line and parameter files and opens UNIX communication pipes for communicating with *Frame Exchange*. As part of the initialization process *Controller Executive* execs the companion *Frame Exchange* process. After completing its initialization *Controller Executive* issues requests to multiple elements of *Exchange Controller*.

▼ Detailed Design

After initialization, *Controller Executive* enters steady state processing and requests services of *Exchange Interface* and *Frame Handler* to determine if any communications have been received from *Frame Exchange* or if any new frames must be provided to *Frame Exchange*, respectively.

In each processing iteration *Controller Executive* checks for an error indication (necessitating termination). Additionally, *Controller Executive* provides signal handling processing for *Exchange Controller*. When a fatal signal is caught or fatal error is encountered, *Controller Executive* attempts a graceful termination for itself, its companion *Frame Exchange*, and the protocol peer *Exchange Controller/Frame Exchange* pair.

In unicast catchup operation, *Exchange Controller* exits when all available frames identified by the missing frame sequence numbers have been sent.

Exchange Interface

At initialization the *Exchange Interface* component establishes communication with the companion *Frame Exchange* process by exchanging a "hand-shake" message. If this initial communication fails, the *Exchange Interface* returns an initialization failure to *Controller Executive* (typically a fatal condition).

During steady state operation of *Exchange Controller*, *Exchange Interface* is periodically requested to poll for frame messages from its companion *Frame Exchange*. When a frame message is discovered *Exchange Interface* initiates appropriate processing to resolve the message. For an instantiation servicing an inbound connection, typically only Data Frame message notifications are received, which causes no further processing. For an instantiation servicing an outbound connection, typically only acknowledgment frame messages are received. The acknowledgment frame message results in a request to *Frame Handler* to mark/log a successful transaction of the identified frame. If a time-out or alert message is received from *Frame Exchange*, a *Controller Executive* method is invoked to gracefully terminate *Exchange Controller*, *Frame Exchange*, and the *Exchange Controller/Frame Exchange* pair of the protocol peer.

Frame Handler

Frame Handler is the largest *Exchange Controller* component. At initialization time *Frame Handler* reads configuration parameters controlling, among other items, the priority scheme for sent protocol frames and opens all *Frame Store* frame sets to be used by *Exchange Controller*. *Frame Handler* categorizes the frame sets it uses as follows:

- reading frame sets—those with frames from the protocol peer
- writing frame sets—for frames created by *Exchange Controller*, which would typically contain only Alert Frames for termination
- polling frame sets—for frames to be sent to the protocol peer
- special logging frame set—for logging the transaction state of frames sent from the polling sets

For traditional unicast initialization, *Frame Handler* attempts to recover from its last execution as follows:

1. When a logging frame set is opened *Frame Handler* determines if the set is empty.
2. If the frame set is empty, *Frame Handler* assumes that a serious failure has occurred, causing the zero content. A serious failure results in a frame message for *Frame Exchange* to request a complete resynchronization of frame sets with the protocol peer.
3. When the logging frame set is not empty *Frame Handler* gets (from the logging set) those frames that were sent to the protocol peer, but not acknowledged as received by the protocol peer.
4. These frames are then resubmitted to *Frame Exchange* (via *Exchange Interface*) for sending to the protocol peer.

For unicast catchup initialization, *Frame Handler* reads the missing sequence number file for frames to be sent.

▼ Detailed Design

During steady state operation an instantiation servicing an outbound connection polls the polling sets for new frames that need to be sent. Frames to be sent are given a send priority based on data time (for Data Frames) and configured transmission policy (for example, LIFO or FIFO) and communicated to *Exchange Interface*. Sent frames are logged as being queued for sending in the special logging frame set.

As part of a graceful termination *Frame Handler* creates an Alert Frame notifying both the local *Frame Exchange* and the protocol peer *Exchange Controller/Frame Exchange* pair that a shutdown should be executed. The Alert Frame is provided to *Exchange Interface* for proper propagation. After providing the Alert Frame, *Frame Handler* closes all open frame sets and returns control to *Controller Executive*.

Input/Processing/Output

Exchange Controller is initiated by either *Connection Originator* or *Connection Manager Server* and is based on whether an outbound or inbound connection is to be serviced, respectively. A companion *Frame Exchange* process is `exec'd` by *Exchange Controller*, and the *Frame Store* frame sets are opened.

Exchange Controller polls *Frame Exchange* for messages and takes action based on the content of the received message. During steady-state operation *Exchange Controller* typically only receives notifications about Data Frames received, which causes no further processing, and acknowledgments for frames sent by *Frame Exchange*. *Exchange Controller* polls frame sets for new frames that need to be sent. Frame to be sent are given a send priority and are communicated to *Frame Exchange*, which affects the transmission. If an Alert Frame or time-out message is received from *Frame Exchange*, termination processing is invoked for the graceful shutdown of the protocol peer processes.

Control

Exchange Controller requires as one of its command-line arguments the value of an open file descriptor to a communications port. For this reason, the process is always initiated by another program: *Connection Originator* for outbound connec-

tions and *Connection Manager Server* for inbound connections. This startup method has the advantage of previously establishing a validated communication port to the protocol peer and diminishes the opportunities for failed startups of the *Exchange Controller/Frame Exchange* pair. After *Exchange Controller* has initialized, the process is controlled via asynchronous events which are either the receipt of a frame message from *Frame Exchange* in *Exchange Interface* or, for traditional unicast operation, the discovery of new frames in polled frame sets by *Frame Handler*. The discovery of new frames results in a request that the frame be sent by *Frame Exchange* to the protocol peer. Frame messages from *Frame Exchange* trigger processing in *Exchange Controller* based on the type and content of the message. In particular:

- If a time-out is received from *Frame Exchange* indicating that the connection to the protocol peer is lost, *Exchange Controller* initiates a termination of *Frame Exchange* and itself.
- If an Alert Frame is received by *Frame Exchange* and communicated to *Exchange Controller*, *Exchange Controller* initiates a termination sequence.

The *Controller Executive* component of *Exchange Controller* contains a signal handler that responds to termination signals. Execution of the signal handler results in an orderly shutdown of the *Exchange Controller/Frame Exchange* pair. The signal handler is invoked when a controlling program wants to terminate *Exchange Controller*, for example, *Data Center Manager* wants to terminate an instance servicing a forwarding connection.

In the absence of a terminating signal or message, *Exchange Controller* executes indefinitely for traditional unicast operation. For unicast catchup operation, *Exchange Controller* exits after the missing frames have been sent and acknowledged.

Interfaces

Exchange Controller has external interfaces with *Frame Exchange*, the Frame Store, and the UNIX file system through the logging library and the missing frame sequence numbers file.

▼ Detailed Design

The interface between *Exchange Controller* and *Frame Exchange* is encapsulated in the processing of *Exchange Interface* and is realized with the UNIX pipe facility, which supports bidirectional communication between the processes. *Exchange Controller* is responsible for the creation of pipes in both directions and provides file descriptors to *Frame Exchange* when it is forked. A data structure called a frame message is used to communicate information over the pipes; both *Frame Exchange* and *Exchange Controller* are capable of creating, sending, and receiving frame messages.

The Frame Store access is encapsulated in the *Frame Handler* component of *Exchange Controller*. The Frame Store is accessed to discover frames that must be sent, to retrieve frames that have been received, and to store the created frames. Access to the frame sets within the Frame Store is via the application program interface provided by *libfs* functions. These functions allow *Frame Handler* to query, retrieve, update, and create data entries. Generally, *Exchange Controller* is passive with respect to the Frame Store, that is, it operates in a read-only mode. The following are exceptions to this rule:

- The logging of frame status is queued for sending and is acknowledged as received. (*Exchange Controller* uses a built-in capability of the Frame Store to handle logging.)
- The creation of Alert Frames, which trigger graceful termination, are saved in the Frame Store.

All components of *Exchange Controller* can use *liblog* to write to a log file to record execution events.

Internally, interfaces between the elements of *Exchange Controller* are almost entirely via function calls. In a few instances data objects are visible between processing objects (global). These items are placed in the header files of the defining processing object/component and are kept to a minimum. When larger data structures are used between functions, access to the structures is by reference, as opposed to by value (passing whole structures).

Error States

Exchange Controller writes program execution information to a log file. The location and name of this file are configurable and are specified in the configuration file provided on the command line when the program is invoked. The log file is an ASCII text file. Each log entry made by *Exchange Controller* specifies the routine that logged the message, and for error conditions, the log number of the message; log numbers for *Exchange Controller* are in the range of 12,000–12,999. Because log messages from other CDS CD-1.1 components are not commingled with *Exchange Controller* log messages, log numbers may be largely ignored. Within the range of allocated numbers there is no scheme to the assignment of values.

Prior to invoking *Exchange Controller* the initiating program establishes a communication port with the protocol peer. If this process fails *Exchange Controller* is not invoked. Although this may appear to be an *Exchange Controller* failure, the actual processing of *Exchange Controller* has not yet started, and therefore this condition is a failure during the processing of the initiating program. This condition is detected by the absence of an *Exchange Controller* log file, that is, if the log file was created (new file time) then *Exchange Controller* began its execution, because log file creation is one of the first actions of the program. If the log file was not created then there was a problem with the invoking program. The exception to this is when a problem is encountered in reading the configuration file for the log file specification, in which case, *Exchange Controller* terminates with diagnostic output going to UNIX standard out and the operating system's *syslog*.

Most failures of *Exchange Controller* are from faulty configuration. In particular the software must be configured as follows prior to program execution:

- The log directory specified in the configuration file exists and is writable by the program.
- The Frame Store configuration file identified in the configuration file is complete and correct.
- All frame sets used by *Exchange Controller* (specified in the configuration file) exist along with their associated frame log files and are described in the Frame Store configuration file.

▼ Detailed Design

- The path to *Frame Exchange* program is correct; *Frame Exchange* exists, and the specified parameter file for *Frame Exchange* is correct. *Exchange Controller* and *Frame Exchange* should use the same configuration file.
- The path to authentication certificate directories exists and is correct; it is contained in the configuration file.

At program initialization *Exchange Controller* and *Frame Exchange* exchange frame messages to communicate that they are ready for execution. If the *Frame Exchange* fails to provide this message then *Exchange Controller* times-out, provides a message in the log file, and exits.

Frame Exchange

The purpose of *Frame Exchange* is to provide reliable low-level processing to move CD-1.1 frames from one protocol peer to another. In particular, *Frame Exchange*, along with its associated *Exchange Controller* attempts to duplicate the content of frame sets in a local Frame Store in a remote Frame Store through the transmission and acknowledgement of frames sent and received. As presented in the discussion of [“Exchange Controller” on page 79](#), *Frame Exchange* always exists as a processing pair with its associated *Exchange Controller*. The processing pair is instantiated once for every connection at the IDC. [Figure 27](#) shows the context of *Frame Exchange*.

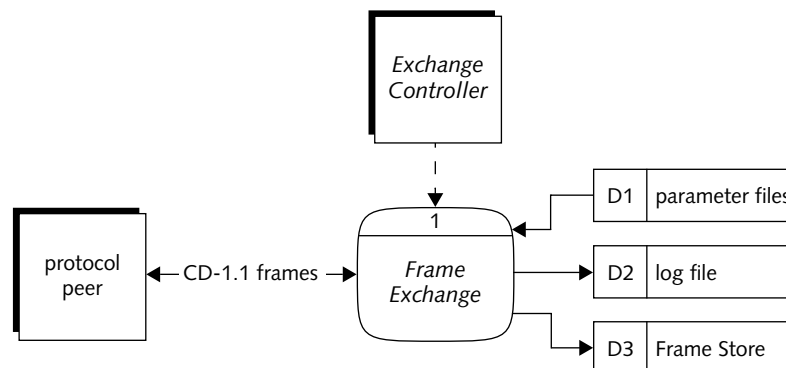


FIGURE 27. FRAME EXCHANGE CONTEXT

Frame Exchange is composed of the following subcomponents:

- *Frame I/O-libfio*
- *Heartbeat*
- *Main Loop*
- *Message Sender*
- *Sender*—one instantiation for each frame set
- *Time Counter*

All components but *Time Counter* are manifest as a processing thread. *Frame I/O* is manifest as a grouping of library/utility type functions. The following paragraphs present the design of each of these subcomponents. [Figure 28](#) shows the interactions of these components.

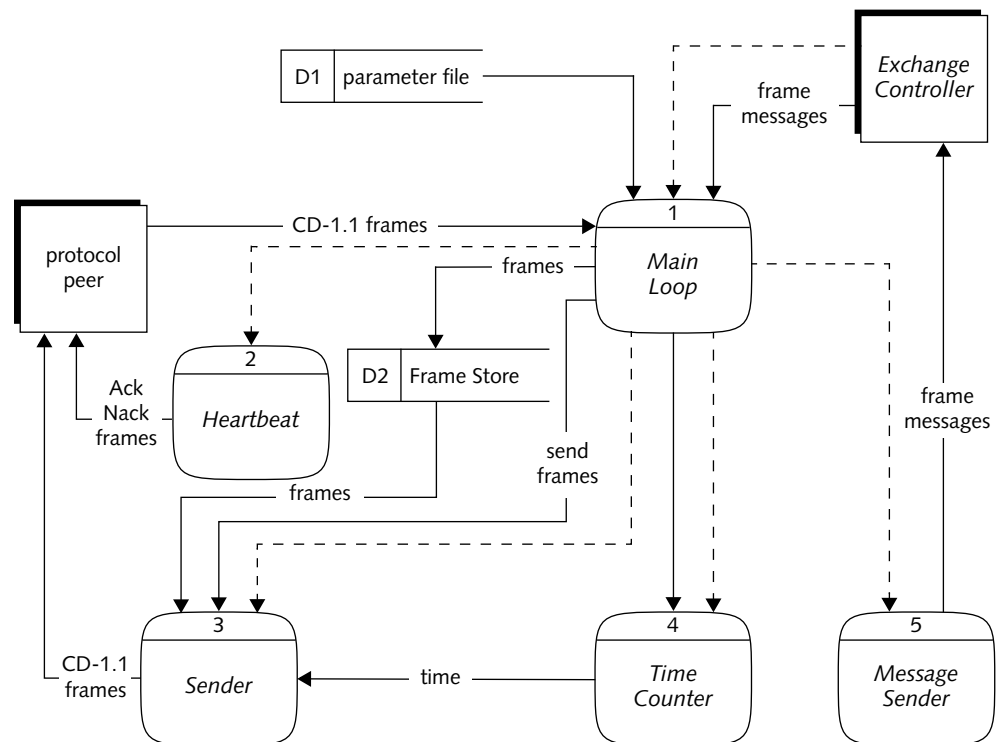


FIGURE 28. FRAME EXCHANGE COMPONENTS

▼ Detailed Design

Main Loop

Main Loop provides the main entry point for the UNIX process (*Frame Exchange*). As its name implies, *Main Loop* contains the central processing frame of the process. *Main Loop* is initiated by *Exchange Controller* to service either an outbound or inbound connection. The UNIX command-line style command to start *Exchange Controller* provides a communication socket file descriptor (to a remote *Frame Exchange*), a communication pipe file descriptor (to the associated *Exchange Controller*), and a parameter file designation. The parameter file supplied to *Frame Exchange* provides run-time configuration values for customizing the execution of the process. The configuration parameter file used by *Frame Exchange* is shared with *Exchange Controller*. Sharing parameter files aids in synchronizing the processing of the two processes. At startup *Frame Exchange* reads configuration values and initializes internal data structures used by sibling processing threads. After successful initialization, *Main Loop* processing instantiates all the required processing threads for *Frame Exchange*.

After all threads are forked, *Main Loop* enters its steady state operating loop where it iterates until a terminating condition is encountered. In each processing iteration *Frame Exchange* checks a timer value to verify that communication to the peer exchange has occurred before the time-out threshold. If the threshold is exceeded the processing terminates processing threads, closes open resources, and exits. Assuming there is no time-out *Main Loop* checks for inbound communication. Input may be received from the peer *Frame Exchange* over an open socket or from *Exchange Controller* over an open pipe. *Main Loop* examines the type of communication/message received and branches appropriately for handling the message.

Time Counter

Time Counter provides an independent execution thread for providing elapsed time counting and is the simplest of the *Frame Exchange* threads. *Time Counter* consists of a processing loop, which updates a time counter for each iteration. As a separate execution thread, the counter provides other processing threads with a concept of elapsed time. The time counter exported by *Time Counter* is protected through the use of mutexes.

Heartbeat

Heartbeat is responsible for sending AckNack messages to a *Frame Exchange* peer. The AckNack message and *Heartbeat* component fulfill two objectives for *Frame Exchange*. First, the AckNack message serves as a keep-alive or *Heartbeat* notification to the peer *Frame Exchange*. In the event that no frames need to be sent between two *Frame Exchanges*, the AckNack message provides assurance to the peer process that the communication line is still open and that the peer process is alive. The second objective is to provide positive feedback regarding the successful receipt of protocol frames. This second objective is accomplished using gap lists, which the *Frame Exchange* uses to track messages sent. One gap list exists for each frame set accessed by *Frame Exchange*.

Heartbeat processing consists of a loop where the processing checks the gap list and *Heartbeat* interval to determine if an AckNack message should be sent. As necessary, the AckNack message is constructed and queued for sending to the peer *Frame Exchange*.

Message Sender

Message Sender is responsible for sending frame messages to *Frame Exchange's* associated *Exchange Controller*. A frame message is a data structure that the *Frame Exchange* and *Exchange Controller* use to communicate. In use, either one of these processes may instantiate a frame message data object, assign data values to its fields, and write the object to the communication pipe. The frame message is an application program entity and should not be confused with CD-1.1 frames.

Message Sender, like the other processing threads, consists of a loop wherein a message queue is polled for messages that must be sent. When messages are discovered on the queue, they are removed and written to the communications pipe. Frame messages are most often placed on the frame message queue by processing in the *Main Loop* component in response to either a frame from the protocol peer or other processing.

▼ Detailed Design

Sender

Sender is instantiated once for each frame set accessed by *Frame Exchange* and is responsible for sending queued frames to the peer *Frame Exchange* over the communication socket. While *Sender* may be instantiated several times, there is only one open socket to the protocol peer. Consequentially, a mutex is used to guard access and use of the socket resource. The *Sender* design is loop based, in which the instantiated thread continues iterating until a terminating condition is encountered. In each iteration *Sender* checks the age of sent frames and checks for new frames that need to be sent. When the age of a frame exceeds a configured threshold, *Sender* resends the frame.

Frame I/O

Frame I/O is a grouping of routines that provides low-level input and output processing needed by other components of *Frame Exchange* for sending and receiving frames. Additionally, *Frame I/O* is constructed as an archive library to allow other processes to take advantage of its processing capabilities.

Processing Lists

In addition to the processing components listed above, an important part of *Frame Exchange* design is its use of lists. Lists are used by *Frame Exchange* to manage frame sending activities. A set of lists is instantiated for each frame set accessed by *Frame Exchange* (with the exception of the message queue). The list set is loosely associated with a *Sender* instantiation. The four lists used by *Frame Exchange* are:

- priority queue
- sent list
- gap list
- message queue

The priority queue is a list of frames to be sent to a protocol peer. Frames are placed in this list by processing in *Main Loop* based on a priority value provided by *Exchange Controller* with the frame. The *Sender* component pulls frames off the top of this list and sends them to the protocol peer. A frame that has been sent is marked by *Sender* as being sent, which effectively places it on the sent list.

The sent list contains those frames that have been sent, but have not yet completed acknowledgement processing. Frames are placed on this list by the *Sender* component, which is also responsible for removing them. A frame is removed from the sent list either because it was acknowledged or was not acknowledged and needed to be resent. (See the following gap list discussion for information on acknowledgement determination.) In each processing iteration the *Sender* component increments the age of the sent list frames and monitors the age with respect to a configured threshold. When a frame has aged beyond the threshold, it is removed from the sent list and placed back on the priority queue to be resent.

The gap list contains entries that identify gaps in frame sequence numbers of frames that were acknowledged by the protocol peer. For example, if frames 1–10 were sent, and only frames 5 and 6 were not acknowledged, the gap list would have one entry identifying 5 as the first missed frame and 7 as the next present frame. When a frame is acknowledged, the *Main Loop* component updates a frame set gap list. For each processing iteration *Sender* checks the frames on its sent list with its gap list. When the gap list indicates that a sent list frame has been acknowledged, *Sender* marks the frame as being acknowledged. *Main Loop* processing examines the sent list for acknowledged entries and creates message queue entries to notify *Exchange Controller* about acknowledged frames. An acknowledged frame then has its sent list entry set by *Main Loop* to show it as available for deletion. *Sender* removes the sent list entries marked for deletion.

The message queue is a list of frame messages sent to *Exchange Controller*. Typically these messages announce the arrival of or the acknowledgement of a frame. The messages are placed on the queue by *Main Loop* processing. *Message Sender* takes messages off the top of the queue and sends them to *Exchange Controller*.

Input/Processing/Output

Frame Exchange requires open file descriptors for I/O as input to *Exchange Controller* and the peer *Frame Exchange* (see [“Main Loop” on page 90](#)). Additionally, *Frame Exchange* accepts command-line style arguments that are read at initialization time to specify run-time configuration options.

During steady state operation (not initialization) three I/O entities are used by *Frame Exchange*:

- communication pipe from *Exchange Controller*
- TCP/IP socket to the peer *Frame Exchange*
- Frame Store

As an input source, the *Exchange Controller* communication pipe is used to provide frame processing requests and notifications. The frame processing requests are used to send a frame. *Frame Exchange* retrieves an identified frame from the Frame Store, and uses the priority value provided by *Exchange Controller* to put the frame into the priority queue for sending to the peer *Frame Exchange*. If an alert notification is received from *Exchange Controller* over this interface, *Frame Exchange* terminates all its processing threads and exits. Output to the communication pipe are frame and alert notifications. A frame notification is provided after successfully receiving and storing a frame from the peer *Frame Exchange*. Alert notifications are provided to notify *Exchange Controller* about terminating conditions, such as the receipt of an Alert Frame, a time-out on *Heartbeat* from the peer *Frame Exchange*, or a broken socket signal (SIGPIPE). Data exchanged across the communication pipe interface are formatted according to a program data structure called a frame message.

Input from and output to the peer *Frame Exchange* occur over a TCP/IP socket. Data exchanged over this interface are CD-1.1 frames. When a frame is received *Frame Exchange* first determines if the frame is an alert or AckNack Frame. Alert frames trigger shutdown processing of *Frame Exchange* and its *Exchange Controller*. AckNack Frames serve two purposes: to provide a heartbeat signal between peer *Frame Exchanges* and to acknowledge frames received. When an AckNack is received, *Frame Exchange* resets its heartbeat timer and then processes the frame's

contents to update the gap list(s) for frames sent. AckNack Frames are not saved in the Frame Store. Any other frame received is stored in the Frame Store and results in sending a frame message to *Exchange Controller*.

The Frame Store is accessed for storing and retrieving CD-1.1 frames. Frames received from the peer *Frame Exchange* are written to the Frame Store. Frames to be sent to the peer *Frame Exchange* are retrieved from the Frame Store (see [“Socket Processing” on page 59](#)). All access to the Frame Store is via the application program interface provided by *libfs*.

Control

Frame Exchange is always initiated by *Exchange Controller*. At startup *Frame Exchange* and *Exchange Controller* coordinate opening the Frame Store frame sets that are accessed by both programs by sending each other frame messages to signal an okay to proceed. After this coordination and after all needed frame sets are opened, *Frame Exchange* instantiates all of its processing threads.

Each thread of *Frame Exchange* executes independently from other threads, including the main thread (though the main retains the ability to cancel/terminate the other threads). While all threads execute independently they all rely on data in a common address space. To protect the fidelity of data and execution of the program, mutexes are used for controlling data access and interaction between the threads. In instances when a common resource is needed a given thread's execution may be suspended because of a mutex lock. A thread suspended in this way resumes execution as soon as the needed mutex is released/unlocked by the holding thread. The coding of *Frame Exchange* is optimized to reduce mutex lock times. Beyond mutex suspended instances, the threads of *Frame Exchange* are cyclic in execution. During each execution cycle, the thread responds to control values, input, and output according to their purpose.

Frame Exchange executes indefinitely in the absence of a terminating condition or communication. The following conditions cause *Frame Exchange* to execute an orderly shutdown and exit:

- An Alert Frame is received from the peer *Frame Exchange*.

▼ Detailed Design

- An Alert Frame message is received from *Exchange Controller*.
- A time-out occurs on receiving an AckNack Frame as a heartbeat from the peer *Frame Exchange*.
- The socket to the peer *Frame Exchange* is broken/closed.
- The pipe to *Exchange Controller* is broken/closed.

In each of the above conditions *Frame Exchange* attempts to notify *Exchange Controller* and peer *Frame Exchange*, as appropriate, about the condition. *Frame Exchange* then terminates all processing threads, closes open frame sets, and exits.

Interfaces

Frame Exchange has external interfaces with *Exchange Controller*, a peer *Frame Exchange*, the Frame Store, and the UNIX file system through the logging library:

- The interface between *Frame Exchange* and *Exchange Controller* is realized with the UNIX pipe facility, which supports bidirectional communication between the processes. A data structure, called a frame message, is used to communicate information over the pipes. Both *Frame Exchange* and *Exchange Controller* are capable of creating, sending, and receiving frame messages.
- An interface to the peer *Frame Exchange* is conducted over a TCP/IP socket and is bidirectional. The messages sent over this interface are frames of the CD-1.1 protocol.
- The Frame Store is accessed to retrieve frames that need to be sent and to store frames that have been received. Access to the frame sets within the Frame Store is via the application program interface provided by *libfs*.
- All threads and components of *Frame Exchange* have the capability to write to a log file for recording execution events. The common *liblog* library is used to write to the file system and to manage file creation/manipulation.

Internally the threads of *Frame Exchange* interface with one another through data in the common address space. When each thread is created it is provided with a pointer to a shared data structure. The scope of data in common for any given thread is dependent on design considerations, so not all threads have access to the same set of data; that is, the same pointer is not provided to all threads. See [“Processing Lists” on page 92](#) for a discussion of the key common address space data entities. A thread communicates and provides data values to other threads by simply setting variables in the shared structure. Access to shared data is controlled by mutexes to avoid collisions between the processing of two or more threads. See [“Control” on page 95](#) for information on how mutexes influence execution control.

Error States

Frame Exchange and *Exchange Controller* share failure modes related to suitability of the operational environment. The following conditions must exist for *Frame Exchange* to execute correctly at startup:

- The log directory specified in the configuration file exists and is writable by the program.
- The Frame Store configuration file identified in the configuration file is complete and correct.
- The frame sets used by *Frame Exchange* (specified in the configuration file) exist along with their associated frame log files; these are the frame sets and frame logs described in the Frame Store configuration file.
- The path to authentication certificate directories exists and is correct and contained in the configuration file.

At program initialization *Exchange Controller* and *Frame Exchange* exchange messages to communicate that they are ready for execution. If *Exchange Controller* fails to provide this message, *Frame Exchange* is suspended in a perpetual wait until it is killed manually.

▼ Detailed Design

After initialization is complete and the program enters steady state operation, all anticipated error conditions are logged to *Frame Exchange*'s log file. The following error conditions may be experienced:

- A broken pipe error occurs when communication is lost to the *Exchange Controller*; *Frame Exchange* exits in this condition.
- A broken connection to peer error occurs when the communication connection to the peer *Frame Exchange* is lost; *Frame Exchange* exits in this condition.
- A peer time-out error occurs when the time threshold for receiving an AckNack Frame from the peer *Frame Exchange* is exceeded; *Frame Exchange* exits in this condition.
- A frame received already accounted for error occurs when *Frame Exchange* determines that a received frame already exists. This error may occur because of processing latency, that is, a frame is resent before an acknowledgement is received; the duplicate frame is ignored.

Multicast Sender

Multicast Sender resides on a multicast data provider. It reads frames from a Frame Store, fragments them into packets, and sends the packets to the multicast group. [Figure 29](#) shows the context for *Multicast Sender*.

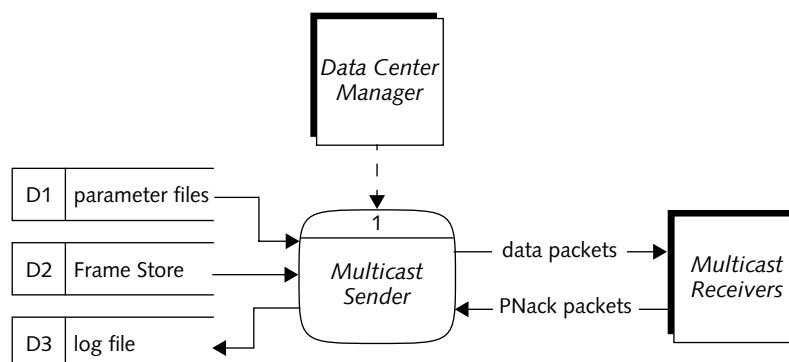


FIGURE 29. MULTICAST SENDER CONTEXT

Multicast Sender listens on a UDP unicast port for PNack packets sent by data consumers requesting missing data packets. If the requested data packets are still available in its buffer, it remulticasts them to the multicast group. If not, it silently ignores the PNack request. *Multicast Sender* maintains a history of its activities in a log file.

Processing

[Figure 30](#) shows data and control flow for *Multicast Sender*. *Multicast Sender* consists of five main objects:

- *MCastProvider*
- *PacketList*
- *MCastFrame*
- *CDS_Socket*
- *CDS_Signal* (not shown)

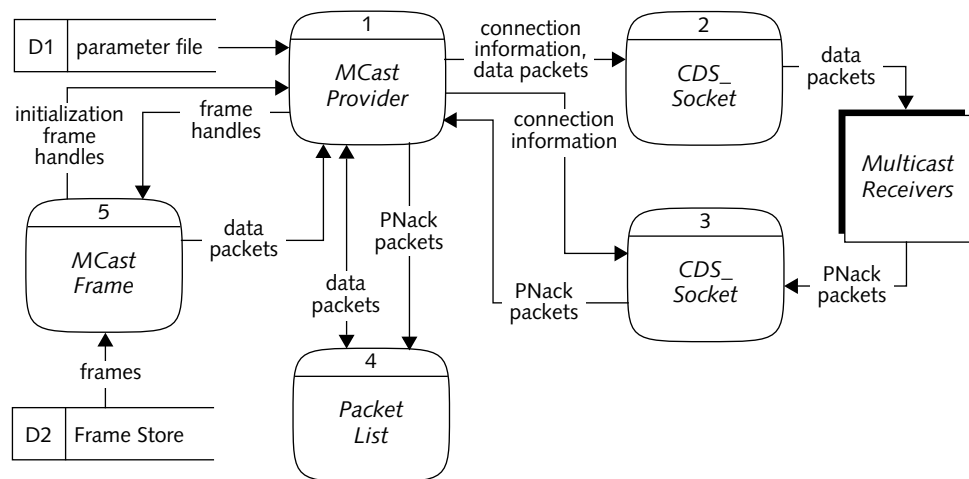


FIGURE 30. MULTICAST SENDER DATA AND CONTROL FLOW

▼ Detailed Design

MCastProvider

MCastProvider is responsible for the following:

- Initialization of the *Multicast Sender* application
- Main processing loop
- Shutdown of the application

MCastProvider is the main processing object for the *Multicast Sender* application. This object controls the timing and execution of all processing methods required to retrieve, fragment and multicast CD-1.1 frames. Processing initiated by this object is divided into five areas:

- Process initialization
- Data input
- Frame fragmentation
- Multicast rate control
- PNack packet processing

MCastProvider is started from the command line or using *Data Center Manager*. Process initialization occurs once, at startup. This processing establishes two socket connections, initializes and creates *PacketList*, opens the Frame Store for reading, and sets up logging and configuration parameters used by the application. Once initialization has completed successfully, *MCastProvider* enters “main loop” processing. If any part of initialization fails, *MCastProvider* terminates.

Data input encompasses two types and sources of data, packetized frames and PNacks. CD-1.1 frames are retrieved from the Frame Store using methods from *MCastFrame*. PNack packets come from the data consumers via a UDP unicast port and are passed directly to *PacketList* for further processing. Receipt of a PNack packet tells *PacketList* that some packets were not received by a data consumer and need to be remulticast.

Frame fragmentation is the process of segmenting a CD-1.1 frame into data packets. Each data packet contains a header and a portion of the CD-1.1 frame. *MCastFrame* is responsible for frame fragmentation. Data packets are stored in *PacketList*.

MCastProvider is responsible for controlling the delivery rate of the packetized frames via multicast socket. At start up, a maximum transmission rate is established via an initialization parameter. *MCastProvider* uses this rate to determine how many packets can be sent over the network for a given period of time. Rate-based flow control attempts to avoid overwhelming the network with traffic and thus reduce collisions and packet loss. As a consequence, packet delivery may fall behind for a period of time. In this case, *MCastProvider* refrains from retrieving new frames until sufficient space is available in *PacketList*.

If *MCastProvider* receives a signal to shutdown, *MCastProvider* begins its shutdown procedure and sends a packetized CD-1.1 Alert Frame via multicast socket notifying the multicast group of an impending shutdown. Once the alert has been sent, sockets are closed, the Frame Store is closed, and *MCastProvider* exits.

PacketList

PacketList performs packet management for data consumers and data providers. For a data provider, *PacketList* is responsible for the following:

- Initializing the *PacketList*
- Storing and retrieving data packets
- Processing PNack packets

As frames are retrieved from the Frame Store, they are fragmented into packets and stored in *PacketList* where they are kept until overwritten by circular buffer management. *PacketList* also maintains the number of times each packet is sent and the last time the packet was sent. These numbers allow *PacketList* to cull out duplicate or premature requests for the packet.

PacketList receives PNack packets from *MCastProvider*, processes them, and if the missing packets are still in the internal buffer, sends them to *MCastProvider* for remulticast. PNack packets may be received from different data consumers identifying the same missing packet. *PacketList* ignores duplicate PNacks and remulticasts a packet only once for a given period.

MCastFrame

MCastFrame performs frame management and transformations between frames and packets for data consumers and data providers. *MCastFrame* is derived from the *CDS_Frame* object. It augments the *CDS_Frame* functionality with methods to perform frame fragmentation and reassembly, and frame verification.

For a data provider, *MCastFrame* is responsible for the following:

- Retrieving frames from the Frame Store in FIFO order.
- Fragmenting frames into packets.

MCastFrame is instantiated by *MCastProvider* for each CD-1.1 frame that is to be retrieved from the Frame Store. *MCastFrame* uses *libfs* to read frames from the Frame Store and fragments them based on the packet size provided at initialization. Constructed packets are provided to *PacketList* for further processing. Once fragmentation is complete, the object is destroyed.

CDS_Socket

CDS_Socket is responsible for the following:

- Creating a socket connection
- Reading from a socket
- Writing to a socket
- Closing socket connection

CDS_Socket contains methods for interfacing with a socket. These methods provide the object user a flexible, full-service interface with a socket. *CDS_Socket* allows a caller to specify the parameters of the socket via calling parameters during initialization. At termination, *CDS_Socket* closes the socket and informs the caller of the termination.

Error States

Multicast Sender is most likely to fail at initialization because of erroneous configuration parameters. The log files usually provide diagnostic information for this type of failure (if *Multicast Sender*'s logging component can be initialized before the failure). The following error conditions may occur at startup:

- The log directory specified in the configuration file does not exist or is not writable by the program.
- The Frame Store configuration file identified in the configuration file is not complete and correct.
- The frame sets used by *Multicast Sender* (specified in the configuration file) and their associated frame log files do not exist.
- The socket for the multicast group or UDP unicast PNack port cannot be opened.

After initialization is complete and the program enters steady state operation, all anticipated error conditions are logged. The following error conditions may be experienced:

- A frame read from the Frame Store fails CRC check.
- The list of missing packets in a PNack exceeds the size of *PacketList*.
- The socket for the multicast group or UDP unicast PNack port is broken/closed. *Multicast Sender* exits after logging the error.
- Memory allocation failure. *Multicast Sender* exits after logging the error.

Multicast Receiver

Multicast Receiver receives multicast data packets, stores them in an internal buffer, reconstructs protocol frames and writes them to a Frame Store. [Figure 31](#) shows the context for *Multicast Receiver*. *Multicast Receiver* is *exec'd* by *Connection Originator* and receives the connection information from its parent.

▼ Detailed Design

Multicast Receiver reads configuration information from parameter files, opens a socket for the multicast group and starts listening to multicast packets. It stores the packets in an internal buffer. If any missing packets are detected, it constructs a PNack packet and sends it to the *Multicast Sender* to request a retransmission of the missing packets. When all the packets for a Frame have been received, the Frame is reassembled and written to the Frame Store. *Multicast Receiver* maintains a log of its activities.

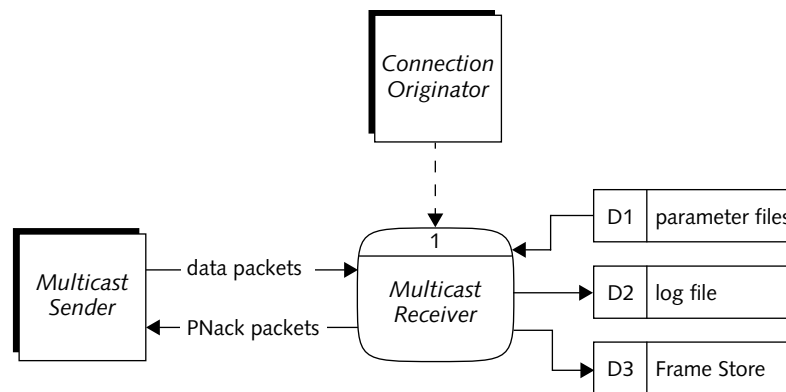


FIGURE 31. MULTICAST RECEIVER CONTEXT

Processing

[Figure 32](#) shows the data and control flow for *Multicast Receiver*. *Multicast Receiver* consists of five main objects:

- *MCastConsumer*
- *PacketList*
- *MCastFrame*
- *CDS_Socket*
- *CDS_Signal* (not shown)

CDS_Socket is described in [“CDS_Socket” on page 102](#).

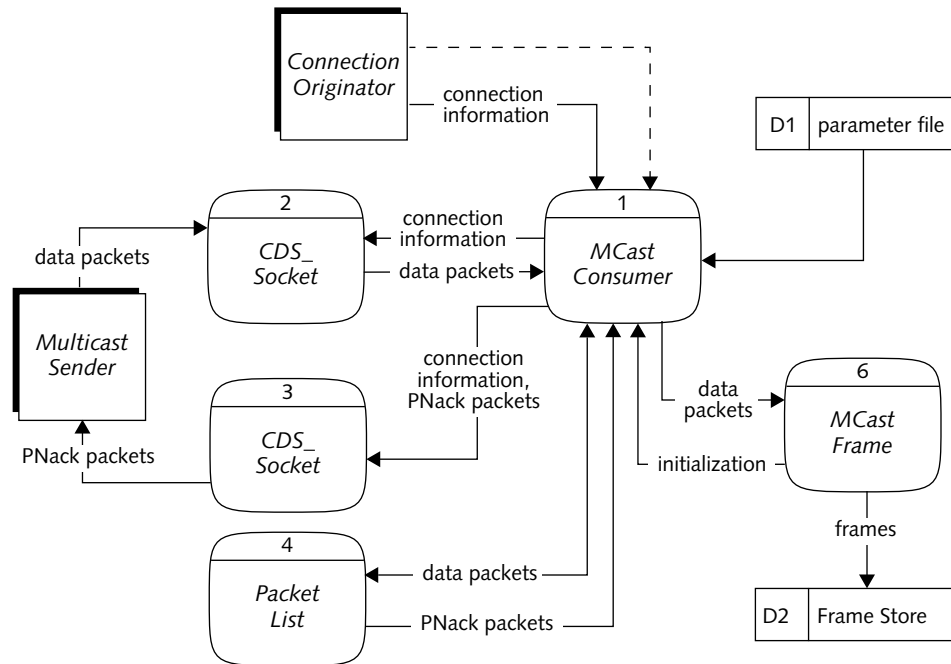


FIGURE 32. MULTICAST RECEIVER DATA AND CONTROL FLOW

MCastConsumer

MCastConsumer is responsible for the following:

- Initialization of the *Multicast Sender* application
- Main processing loop
- Shutdown of the application

MCastConsumer is the main processing object for *Multicast Receiver*. This object controls the timing and execution of all the processing methods required to receive and store multicast CD-1.1 frames. The processing initiated by this object is divided into four areas:

- Process initialization
- Data input

▼ Detailed Design

- Packet assembly and storage
- Obtain missing packets

MCastConsumer is started by *Connection Originator* after CD-1.1 connection procedures are complete. Once started, *Connection Originator* exits and *Data Center Manager* monitors the status of *MCastConsumer*. If *MCastConsumer* exits, *Data Center Manager* starts *Connection Originator* to reestablish the connection and perform a restart of *MCastConsumer*.

Process initialization occurs once at startup. This processing establishes two socket connections, initializes and creates *PacketList*, opens the frame store for writing, and sets up logging and configuration parameters used by the application. Once this processing is successfully completed, *MCastConsumer* enters the main processing loop. If initialization fails, *MCastConsumer* exits.

Data input encompasses the processing required to receive and store data packets sent by a provider. Data packets are received via a UDP socket and stored in *PacketList* where they are kept for further processing.

If data packets are not received, *MCastConsumer* asks the data provider to resend missing packets by sending a PNack packet. PNack packets are created by *PacketList* and are sent to the data provider via the UDP unicast socket. Formats for data and PNack packets are specified in ["Multicast Protocol" on page 135](#). *PacketList* maintains the last time and the number of times a PNack packet was sent. If a missing packet does not arrive, the missing packet is ignored along with other packets that would have potentially been used to reassemble a frame.

MCastConsumer exits if the data provider cannot provide packets. This condition is realized by one of two events. The first is receipt of an Alert Frame from the data provider. The alert tells *MCastConsumer* that the data provider is exiting and allows for a graceful shutdown. The second is a lack of input for a user-defined length of time.

PacketList

PacketList performs packet management for both a data consumer and a data provider. For a data consumer, *PacketList* is responsible for the following:

- Initializing the packet list
- Storing and retrieving packets
- Verification of packets
- Providing a container of PNack packets to *MCastConsumer*

PacketList is charged with the management and storage of two primary data elements between the data provider and the data consumer, data packets and PNack packets. Data packets are fragments of CD-1.1 frames. Each of the fragments contain a header to assist the consumer in reassembling the packets into a complete frame. For each packet a consumer is missing, a PNack packet is generated and sent to *MCastConsumer* to be sent to the data provider.

Data packets are stored as they arrive from the data provider. After a configurable amount of time, *PacketList* checks for missing packets among those previously received and generates a PNack packet for the missing packets. *PacketList* increments the number of requests generated for each packet and the last time a request was sent. This gives *PacketList* the ability to wait a sufficient amount of time for a response from the data provider before a retry.

MCastFrame

MCastFrame is responsible for reassembling packets into CD-1.1 frames, verifying frames, and storing frames in the Frame Store. *MCastFrame* is derived from the *CDS_Frame* object. It augments the *CDS_Frame* functionality with methods to perform frame fragmentation, frame reassembly, and frame verification.

For a data consumer, *MCastFrame* is responsible for the following:

- Reassembling protocol frames from packets
- Verification of reassembled frames
- Writing reassembled frames to the Frame Store

▼ Detailed Design

MCastFrame objects are instantiated by *MCastConsumer* when a new frame is to be assembled. *MCastFrame* is provided the packets that make up the CD-1.1 frame. The packets are stripped of the packet header and placed in contiguous memory. The new reassembled frame is then verified via its included checksum. Once verified, the frame is written to the Frame Store using *libfs* and the object is destroyed.

Error States

Erroneous configuration parameters are the most likely cause of *Multicast Receiver* failure at initialization. The log files usually provide diagnostic information for this type of failure (if *Multicast Receiver's* logging component can be initialized before the failure). The following error conditions may occur at startup:

- The log directory specified in the configuration file does not exist or is not writable by the program.
- The Frame Store configuration file identified in the configuration file is not complete and correct.
- The frame sets used by *Multicast Receiver* (specified in the configuration file) and their associated frame log files do not exist.
- The socket for the multicast group or UDP unicast PNack port cannot be opened.

After initialization is complete and the program enters steady state operation, all anticipated error conditions are logged. The following error conditions may be experienced:

- Reassembled frame fails CRC check.
- Frame is discarded due to missing packets.
- The socket for the multicast group or UDP unicast PNack port is broken/closed. *Multicast Receiver* exits after logging the error.
- Memory allocation failure. *Multicast Receiver* exits after logging the error.

Missing Frame Detector

The purpose of the *Missing Frame Detector* is to evaluate a Frame Store frame set for sequence number gaps. The context for *Missing Frame Detector* is presented in [Figure 33](#). *Missing Frame Detector* is started periodically by *Data Center Manager*. If frames are missing from the sequence, *Missing Frame Detector* writes the list to a file.

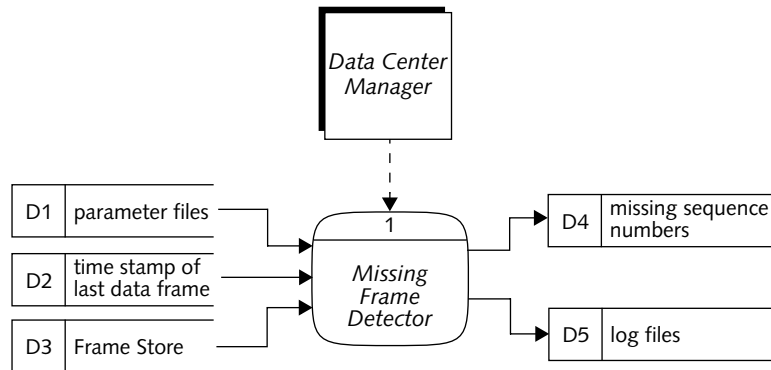


FIGURE 33. MISSING FRAME DETECTOR CONTEXT

Processing

[Figure 34](#) shows the data and control flow for *Missing Frame Detector*. *Missing Frame Detector* consists of two processing components:

- *ListEval*
- *FrameStoreObjects*

▼ Detailed Design

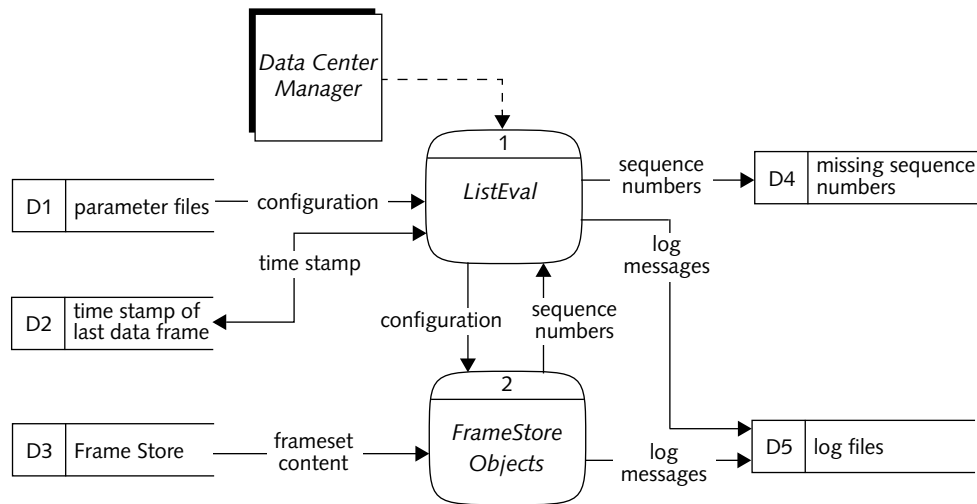


FIGURE 34. MISSING FRAME DETECTOR DATA AND CONTROL FLOW

ListEval

ListEval is the main subcomponent. *ListEval* coordinates initialization of the other program components. After initialization, processing takes place to assess the sequence number gaps in a configured frame set. When one or more gaps are discovered, the process writes the missing sequences to an output file of a configured name. Additionally, the time of the last Data Frame before the first gap is saved to durable store. Completing this task, the process exits, providing a return code to indicate that a missing sequence numbers file was or was not written.

Each time the process is invoked, the above processing is repeated with the following exception. On subsequent invocations, the time of the last Data Frame before the first gap of the immediately proceeding execution is retrieved. This time is used as a search seed time for the subject frame set. In this way *ListEval* can avoid unnecessarily searching portions of the frame set known to be free of sequence number gaps, while still ensuring that an accurate representation of missing sequence numbers for the frame set is captured in the output file.

ListEval interacts with *FrameStoreObjects* to invoke methods for accessing the Frame Store.

FrameStoreObjects

FrameStoreObjects is actually a group of processing objects for performing operations on the Frame Store. This group includes:

- *CDS FrameStore*
- *CDS FrameSet*
- *CDS Frame*

Together these elements support the ability to open and close a Frame Store, open and close a frame set, and request information about the contents of a frame set, for example, query for the list of frames in a frame set. These processing objects are common re-use objects in *CDS CD-1.1*. Their detailed description is not provided here.

Input/Processing/Output

Missing Frame Detector takes its input from the contents of a Frame Store frame set. A frame set is polled for information about its content, in particular the list of frames it contains. The frame list is then processed to determine where there are gaps. A record is constructed for each sequence number gap and written to a file. If multiple frame sets are evaluated by *Missing Frame Detector*, a separate output file is produced for each frame set.

Missing Frame Detector logs messages to an ASCII text log file. The location and name of this file are specified in the configuration file. Log messages note significant processing events and capture error information. Each log entry specifies the routine that logged the message along with the text of the message.

▼ Detailed Design

Control

Missing Frame Detector may be started from the UNIX command line. In an operational environment, invocation will come from an external controlling process such as *Data Center Manager* or UNIX *cron*. When *Missing Frame Detector* executes, it performs its evaluation and exits. Execution should occur on a periodic basis so that missing sequence number files reflect the current state of the Frame Store. Reading the content of a frame set is a time consuming process. For this reason, a balance must be struck between currency and timeliness of output files and utilization of computational resources. That is, *Missing Frame Detector* should execute frequently enough that gap lists reflect reasonably current conditions but not so frequently that a constant list refresh results.

Interfaces

Missing Frame Detector has external interfaces to the Frame Store, the UNIX file system, and to a startup control program. The interface to the Frame Store is encapsulated in *FrameStoreObjects*. At the core of *FrameStoreObjects* is the *libfs* which provides function call methods for performing operations. This interface supports access to open and close a frame set and for polling and reading a frame set's content. *Missing Frame Detector* is passive with respect to Frame Store content. That is, the frame sets are read, but not written.

Both components of *Missing Frame Detector* log messages to a log file. The logging library *liblog*, is at the core of message logging and the interface to the log file.

Internally, interfaces between *Missing Frame Detector* components are almost entirely via function calls. In a few instances, data objects are visible between processing objects (global). These items are placed in the header files of the defining component and are kept to a minimum.

Error States

Missing Frame Detector writes program execution and error information to a log file. Most failures of *Missing Frame Detector* are from faulty configuration. The following must be configured prior to program execution:

- The log directory specified in the configuration file exists and is writable by the program.
- The Frame Store configuration file identified in the configuration file is complete and correct
- All frame sets used by *Missing Frame Detector* (specified in the configuration file) exist along with their associated frame log files and are described in the Frame Store configuration file. If *Missing Frame Detector* is configured to evaluate multiple frame sets and an error occurs accessing a frame set, the process attempts to continue to evaluate other frame sets despite the error.

Data Parser

Data Parser reads continuous time-series data as Data Frames from a Frame Store and writes the enclosed channel data to the Data Management System. This data reformatting operation results in time-series data being available through the Data Management System for the automatic and interactive time-series processing applications. The context of *Data Parser* is shown in [Figure 35](#). This figure illustrates that *Data Parser* provides the coupling between other components of the CDS CD-1.1 and the automatic and interactive processing systems.

An ancillary purpose of *Data Parser* is to provide quality control of the continuous data stream. In particular, because *Data Parser* examines the contents of the data stream, it is a natural process to handle the data signature verification of Channel Subframes, which confirms the authenticity of the data.

▼ Detailed Design

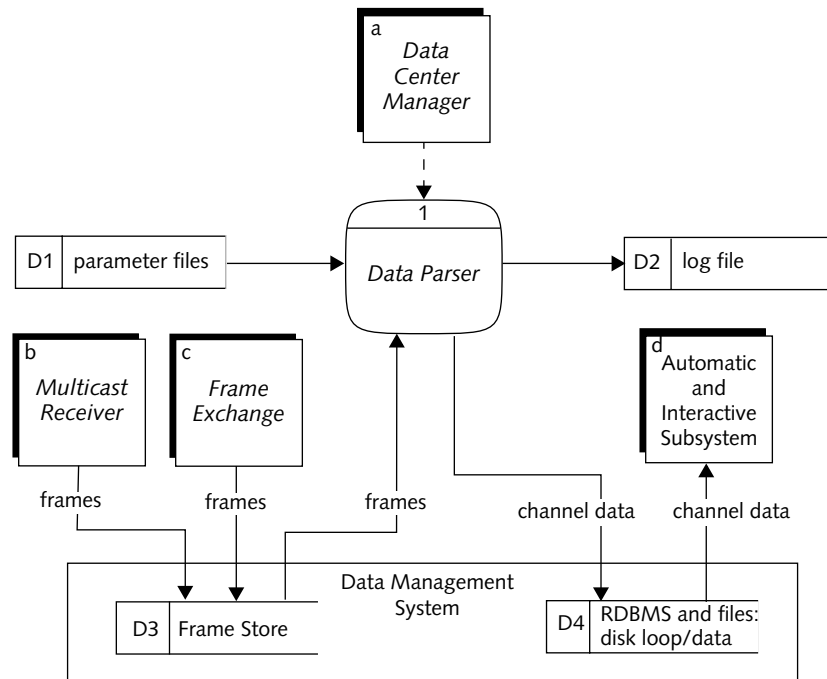


FIGURE 35. DATA PARSER CONTEXT

[Figure 36](#) shows a detailed data flow within *Data Parser*. *Data Parser* has three principal components:

- *DLParse Exec*
- *Process Loop*
- *Process Frame*

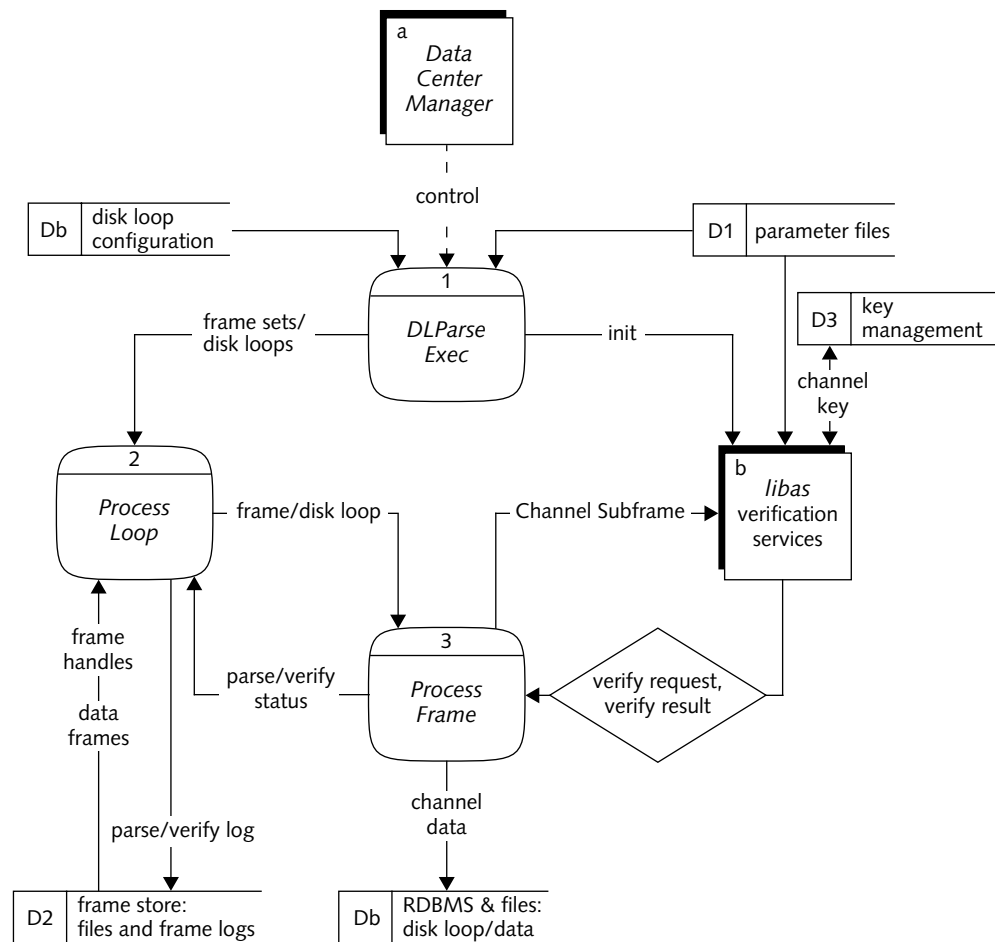


FIGURE 36. DATA PARSER DATA FLOW

DLParse Exec

DLParse Exec is the entry point for the UNIX process that makes up the *Data Parser*. *Data Parser* is intended to run unattended and indefinitely in the background. In a typical operation it is monitored by a *Data Center Manager* process that restarts *Data Parser*, as needed, in the event of a termination. *DLParse Exec* performs the following processing:

▼ Detailed Design

1. Read and parse command-line arguments. The command line is formatted using *libpar* syntax.
2. Initialize logging services. Logging services are provided by *liblog*. All components use the logging services, which is omitted from the figure for simplicity.
3. Initialize signal handling. *Data Parser* is usually terminated by receiving a signal. Signal handling assures that shutdown is handled cleanly. All transactions with the DBMS are completed, and all persistent states are written to the file system.
4. Initialize the verification services. The input time-series data may be signed at the sensor to verify its authenticity. *Data Parser* can be configured to verify the data authenticity. This step sets up the verification services provided by *libas*. *Data Parser* only verifies the signed sensor data (that is the Channel Subframes). This operation is distinct from verifying the signature on the frames. *Data Parser* does not check that the frame signatures are valid.
5. Associate acquisition sites with frame sets. Each acquisition site is named and consists of one or more channels. Time-series data for each channel are managed in a disk loop. A disk loop is a collection of disk files logically ordered in a time-series loop. When new data arrive, the oldest data are removed to make room for the newest data. Thus, every acquisition site is associated with a collection of one or more disk loops. Similarly, a frame set is a data collection consisting of all of the channel data from an acquisition system (see [“libfs” on page 128](#)). Accordingly each input frame set is associated with an acquisition site, and correspondingly, one or more disk loops.

In addition to each input frame set, there are up to two output frame sets for each acquisition site. These output frame sets do not contain any frame data, but instead reference the input frame data. The purpose of these frame sets is (1) to provide status logging regarding the success or failure of the parsing process, and (2) to provide status logging regarding the signature verification processing (if verification is desired).

6. Open the Frame Store and input/output frame sets. For each data provider site, this step initializes the Frame Store reading functions and verifies that the input and output frame sets exist and are accessible. After opening each frame set the last frame processed for each frame set is read from a persistent file.
7. Open the RDBMS and read the disk loop configuration for the acquisition sites. For each acquisition site, query the database and extract all of the relevant information about disk loops related to that acquisition site. disk loops are preallocated and each channel must be configured for processing. Channel configuration includes specifying how the channel data are encoded and whether or not they are authenticated.
8. Pass each frame set/data provider site/disk loops association to *Process Loop*. After *DLParse Exec* initial processing is completed *Process Loop* takes over. No return from *Process Loop* is needed, and the process must be terminated by sending it a signal.

Process Loop

Process Loop is essentially a loop over all input frame set/acquisition site pairs. *Process Loop* performs the following processing steps during each loop:

1. Poll for newly arrived Data Frames. Polling is controlled by input parameters and is intended to be relatively infrequent (about every 30 seconds to a few minutes). The result is that for each frame set several frames are available for processing after each polling cycle. The maximum number of frames to be processed from a frame set in a polling cycle is also a configurable parameter. In this way a single station, which may be sending older data very quickly, cannot block other stations that are sending recent, up-to-date data.
2. For each new frame encountered, read the frame data and invoke *Process Frame*. The polling for new frames results in a collection of frame handles that include references to the actual frames stored in the Frame Store. For each frame handle *Process Loop* reads the frame data and passes the data, along with the acquisition and diskloop information, to

▼ Detailed Design

Process Frame. After processing each frame *Process Frame* returns to *Process Loop* with an overall success/failure status indicator (via return value). For each associated channel, an indicator of whether or not that channel was (1) successfully parsed and (2) successfully verified is also returned.

The status of parsing and verification are written to output frame sets associated with each acquisition site. The parsing status output frame set contains a reference to the input frame and the list of channels that were successfully parsed. The verification status output frame set contains a reference to the input frame and the list of channels that failed authentication. A channel may be successfully parsed, but fail authentication.

3. Periodically flush parsed channel and output frame set data. To reduce the impact of *Data Parser* on system resources, especially the DBMS, transactions to the DBMS are buffered. Each channel's time-series data are written to the file system as they are parsed, but inserting or updating the corresponding DBMS (**wfdisc**) record is deferred. Typically, these database records are flushed at least every 1–2 minutes. If a lot of data are arriving rapidly, this interval is increased to keep the size of the data buffers manageable.

Writing the output frame sets is coordinated with the output to the DBMS. In this way, the output frame sets accurately reflect the data that are successfully parsed, inserted into the DBMS, and made visible to the other processing applications.

Associated with the flushing of the data, *Process Loop* writes state information to nonvolatile disk file storage. The purpose of this operation is to record the sequence number of the last record successfully parsed and loaded into the DBMS. If *Data Parser* is terminated, it can be restarted from its most recent, successful operation.

Process Frame

Process Frame processes the contents of a single Data Frame as provided by a frame handle from *Process Loop*. It handles verification, frame decoding, and output of the channel data to the file system. Upon entry *Process Frame* is provided the frame handle and its associated data provider and disk loop information. *Process Frame* performs the following processing steps:

1. Unpack the frame. On input a frame is a string of bytes and must be unpacked to be interpreted. Unpacking services are provided through *libcdo*.
2. Parse each channel in the frame. Each channel in the frame processing involves (1) verifying the signature, (2) converting the data to its output format, (3) writing the data to the corresponding disk loop file, and (4) updating the corresponding **wfdisc** entry. If a channel fails conversion or writing to the disk loop file it is logged, but processing continues unless the frame itself is corrupt. If a channel fails verification the failure is recorded for later logging.

Input

Data Parser takes input from the following files and database objects:

- Parameters and parameter files. *Data Parser* uses *libpar* to parse parameters. The organization of parameters into parameter files is site specific and discussed in [\[IDC6.5.18\]](#).
- DBMS tables. The following DBMS tables are read: **affiliation**, **dlfile**, **instrument**, **site**, **sitechan**, **sensor**, **wfconv**, **wfdisc**, and **wfproto** (a view on **sitechan**, **sensor**, **instrument**, and **wfconv**).
- Frame Store. *Data Parser* processes one or more input frame sets, which contain Data Frames from which to extract the channel time-series data.

▼ Detailed Design

- Last frame processed files. *Data Parser* records the last frame processed into a small file for each data provider site. Upon process startup, it uses this file to determine the sequence number of the next frame to be parsed.

Output

Data Parser writes to the following files and DBMS objects:

- DBMS tables **wfdisc**. Time-series data files are the files associated with the **wfdisc** table (commonly called ".w" files), which hold the time-series data.
- Frame Store. For each input frame set, *Data Parser* writes up to two output frame sets. One of these frame sets writes one log entry for the parsing status of each input Data Frame. This output frame records the channels that are successfully parsed as well as a reference to the original input frame. The other of these frame sets writes one log entry for the authentication status of each input Data Frame in which one or more channels fail verification. This output frame records only the channels that fail authentication. If all channels are successfully verified, then no authentication status frame is written.
- Last frame processed files. These files contain the sequence number of the last successfully processed Data Frame.
- Log files. *Data Parser* logs its status to log files using *liblog*.

Control

Data Parser is started by the *Data Center Manager*. It continues to run indefinitely until it either encounters an error or receives a signal. At startup *Data Parser* closes "standard" input and output streams and restricts status output to log files as written by logging services (*liblog*). For debugging, *Data Parser* can be configured to process a specific set of frames and stop.

Interfaces

Data Parser exchanges data with the file system, the Frame Store, and the DBMS through the following library interfaces:

- Parameters and configuration data are exchanged through *libpar*.
- Frame Store data are exchanged through *libfs* and *libframelog*.
- Frames are interpreted through *libcdo*.
- Access to the authentication data (public keys) is provided through *libas* and *OpenSSL*.
- DBMS data are exchanged through *libgdi*.
- Time-series data are decoded using *libwio*.
- Time-series (channel) data are written to the file system by functions specific to *Data Parser*.
- *Data Parser* logs are managed through *liblog*.
- The current state file is written to the file system by functions specific to *Data Parser*.

Within *Data Parser*, acquisition site and frame set-related data are exchanged through function calls. Configuration parameters that must be shared between files are stored as globals.

Error States

After *Data Parser* has successfully started it should continue to run indefinitely. Most errors that cause it to fail are encountered immediately at startup and can be diagnosed by examining the log files. The following is a partial list of startup failure modes:

- Improper configuration (parameter values), including missing parameter values, which are noted in the log files.
- The database is inaccessible or improperly specified.

▼ Detailed Design

- Frame sets cannot be opened; invalid Frame Store configuration. If any frame set cannot be opened *Data Parser* continues to process those frame sets that can be opened. However, *Data Parser* terminates when it cannot process any data provider sites.
- Improper configuration of database tables. All channels to be parsed must be configured in the DBMS. Configuration includes setting up data decoding information and allocating disk loops. If any one channel for a station is improperly configured, or if disk loop files are not allocated, *Data Parser* does not process any data it finds from the offending channel. It continues, however, to process the other channels. The log file indicates whenever an unknown channel is encountered in the Data Frame, and configuration problems are detected by observing the log.
- Authentication services cannot be initialized (only if authentication is enabled). Some errors in the authentication configuration are detected initially, and *Data Parser* exits immediately upon encountering them. However, some errors are not detected by the authentication services library until signatures are verified. A separate authentication log contains errors specific to the authentication services. Normally this log is empty, but if certificates are discovered to be missing they are logged there.
- If any channel fails parsing *Data Parser* ignores that channel and continues with the remaining channels in the frame.

If the database connection fails during operation *Data Parser* exits.

Frame Store Stager

The *Frame Store Stager* program is the interface between the *CDS CD-1.1* and the Archiving Subsystem. *Frame Store Stager* monitors active Frame Stores and moves inactive Frame Store files into a staging area in the file system. This staging area is where the Archiving Subsystem acquires Frame Store files. Once *Frame Store Stager* has moved Frame Store files into the staging area, it writes the results of these

Frame Store file manipulations to the DBMS to be used by the Archiving Sub-system to determine which files to archive. [Figure 37](#) shows the *Frame Store Stager* context.

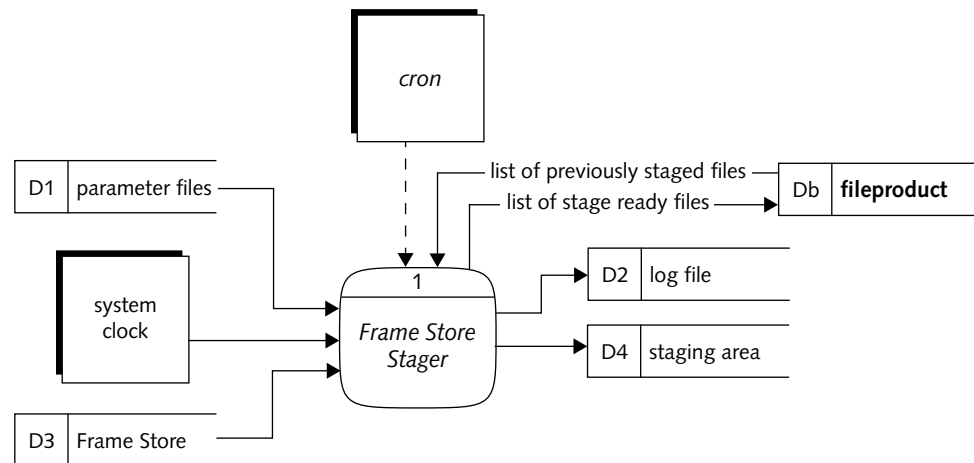


FIGURE 37. FRAME STORE STAGER CONTEXT

Input/Processing/Output

Input for environment initialization occurs when *Frame Store Stager* is started from parameterized data in par files. These data include:

- logging setup data
- active frame set list
- archiving interval
- staging area directory

Frame Store Stager gets the current system time with *libstdtime* library functions. It then compares the current time and the start time for each Frame Store file. When a file start time is earlier than the current time minus the archiving interval, *Frame Store Stager* moves the file to a staging area directory specified by the user. *Frame Store Stager* repeats this process for each frame set in the frame set list. *Frame Store*

▼ Detailed Design

Stager then notifies the Archiving Subsystem that these files are available for archiving. This is accomplished using *libfileproduct* library functions to write file data to the database. After the database update is complete, *Frame Store Stager's* processing is complete and the process terminates.

[Figure 38](#) shows the data flow for *Frame Store Stager* processing and interface components.

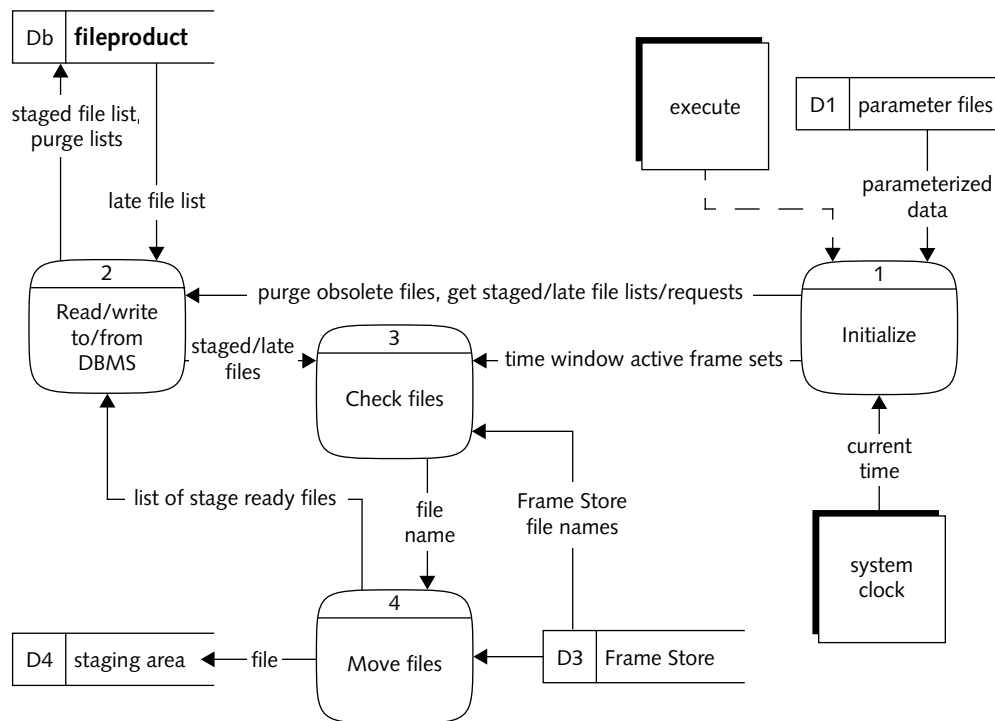


FIGURE 38. FRAME STORE STAGER DATA FLOW

Control

Frame Store Stager is started with the UNIX utility *cron* based on an interval specified by the user. Using *cron*, *Frame Store Stager* runs automatically at specified intervals without user interaction. *Frame Store Stager* may also be executed manually from the UNIX command line at any interval deemed appropriate.

Once the program is started, configuration data determines how *Frame Store Stager* processes Frame Store files. This data controls where data are moved and which Frame Store files are available for archiving.

Interfaces

Frame Store Stager is the interface between the *CDS CD-1.1* and the Archive Subsystem. This interface is indirect and utilizes the database and a predetermined UNIX file system directory as the interface medium. *Frame Store Stager* performs its processing on each of the specified frame sets, moving candidate files to the staging directory. It then generates a **fileproduct** record for each file and stores these in the DBMS. *Frame Store Stager* then updates its local databases. These updates are used by *Frame Store Stager* to determine if a file was previously staged, requiring an alteration of a file name. When the archiver runs, it uses the **fileproduct** records to find and archive staged files.

The interface to Frame Store is through UNIX system calls and calls to *libfs*. Actual manipulation (movement) of the Frame Store files does not require *libfs*, but any data extraction (last write time, volumes, and so forth) uses *libfs* functions. In this way, *Frame Store Stager* does not require any knowledge of the Frame Store file structure or the Frame Store itself.

Frame Store Stager has an interface to user defined par files containing the required parameter definitions. *libpar* library functions are used to access par files.

An interface to log files is used by *Frame Store Stager* to log significant processing events and error conditions. Library functions of *liblog* are used to create, manage, and write messages to these log files.

Error States

Because *Frame Store Stager* uses the UNIX file system to move and process Frame Store files, it is most likely to fail when creating, moving, or deleting Frame Store files. The most probable cause of such a failure is improper file or directory permissions. If proper permissions do not exist, *Frame Store Stager* fails to create staging or logging directories, to open existing directories, or to manipulate Frame Store

▼ Detailed Design

files. In the event of such a failure, *Frame Store Stager* generates an error message and exits. For each fatal error, an error message is created using both *syslog* and *liblog*. These messages give the point of failure and the error code (if any) for the failure.

Frame Store Stager can also fail if it cannot establish a connection to the database. This error also generates error messages from both *liblog* and *syslog*. Because the database connection is essential, *Frame Store Stager* has no other recourse but to exit.

Protocol Checker

The purpose of *Protocol Checker* is to verify that frames in a given frame set are formatted in accordance with the CD-1.1 protocol. The application retrieves Data Frames from a frame set specified by the user and checks data elements in the frame to ensure that they are within the limits specified by the CD-1.1 protocol. This is useful during application development as an indicator of the software's ability to generate and store Data Frames or as a debugging tool if problems arise in the field. *Protocol Checker* is a stand-alone application and can be run on either an active or an inactive frame set. [Figure 39](#) shows the context of *Protocol Checker*.

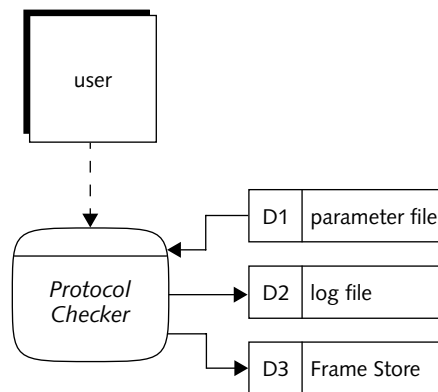


Figure 39. Protocol Checker Context

Input/Processing/Output

Protocol Checker requires a number of inputs to function correctly. These inputs are provided to *Protocol Checker* as either command-line arguments or par file elements. Input parameters identify the configuration of the Frame Store, the frame set of interest, and other parameters that specify how *Protocol Checker* executes.

Another par file is used to input initial values for logging messages to log files. These values include the log filename, log file location, number of log files, log threshold, and the priority of each of the log classes supported.

After the processing parameters are established, *Protocol Checker* uses *libfs* to extract Data Frames from the frame set. *Protocol Checker* then uses *libcdo* routines to check the frame for adherence to the CD-1.1 protocol.

Output from the process is generated as each part of the frame is checked. Messages indicating the value of frame data elements and whether or not the data are within prescribed limits, are generated using *liblog* routines. Messages describing frames are produced in ASCII and are logged to a log file for review by the operator.

Control

Protocol Checker is started by using the command line. The user specifies the executable name followed by the required input parameters. The application reads the parametric data to determine which frame set to parse and which Data Frames to parse. By default, *Protocol Checker* is nonterminating, meaning it runs forever unless the terminate flag is set to true or the program receives a SIGTERM (Ctrl-C) from the user.

Interfaces

Protocol Checker has two major interfaces. The first interface is between *Protocol Checker* and a frame set. For this interface, *Protocol Checker* uses routines provided by *libfs*. These routines are responsible for opening the frame set, retrieving desired frames, and closing the frame set.

▼ Detailed Design

The second interface is between *Protocol Checker* and the files where the results of the verification are stored. *Protocol Checker* uses routines provided by *liblog* for this interface. These routines create log files, write results to log files, and close log files.

Error States

Protocol Checker can only function based on input from the user. Because of this, the most common error state is one that arises from improper user input. These errors occur when the user specifies a nonexistent frame set or a nonexistent par file. In both cases, the program prints an error string to *stderr* and exits normally. This error state is avoided by input of the correct data.

libfs

CDS CD-1.1 processes access the Frame Store via the Frame Store library *libfs*. The Frame Store satisfies all the high-level requirements that pertain to the storage and retrieval of CD-1.1 frames. The Frame Store and the *libfs* relationship to it are described in the following paragraphs.

The Frame Store provides durable storage for CD-1.1 frames. The Frame Store is a dynamic entity: it accepts new CD-1.1 frames for storage, retrieves requested frames, and deletes the oldest frames (after optional archiving). The Frame Store is comprised of the Frame Store files (extension *.fs*) and Index files (extension *.flog*). Index files allow efficient retrieval by insert time, which is the most common type of retrieval. The Frame Store files have the following properties:

- they have unique names
- they are dynamically allocated
- they append new CD-1.1 frames
- they retain data signatures
- they store non-data CD-1.1 frame types
- they are self-describing
- they are archive-ready without further processing

The Frame Store is described to *libfs* by a parameter file that uses the par table feature provided by *libpar*. The parameter file is shared by all *CDS CD-1.1* applications that access the Frame Store. The parameter file assigns each frame set a unique name and identifier, and assigns other attributes such as the directory for the Frame Store files and the size of each file (both in time duration and expected number of bytes).

The Frame Store is subdivided by frame sets. A frame set is a collection of frames from the single data provider or creator. Examples of frame sets are seismic and other monitoring arrays, individual channels of time-series data, and participants capable of creating command and control frames. In the latter case, an NDC that issues command and control frames is considered the creator of those frames, and the command and control frames are assigned to their own frame set.

The *libfs* library presents the Frame Store to applications as a file system object. The library interface includes access functions to open, close, read, write, and search the Frame Store. The library reads and parses the Frame Store parameter file when the open function is called. The Frame Store *open* function returns a handle to the application; subsequent calls to the library provide this handle as an argument. The Frame Store *close* function releases all resources allocated for the Frame Store. Between open and close functions the calling application may access any of the *libfs* functions. All library functions are transactional; this feature allows simultaneous access to the Frame Store by several processes.

Input/Processing/Output

The inputs to *libfs* are as follows:

- Frame Store parameter file (See the *frameStore(5)* man page for a description.)
- Frame Store files (*.fs*) (These disk files contain intact CD-1.1 frames. See ["Interfaces" on page 60.](#))
- Index files (*.flog*) (These disk files contain an index into the Frame Store files. See ["Interfaces" on page 60.](#))

▼ Detailed Design

- calling arguments supplied by the applications (See the *libfs(3)* man page for arguments.)
- host system time (The library obtains current time values from the host system.)

Processing of *libfs* follows a procedural-based library model: the control flow remains with the calling application, and each *libfs* function executes to completion and returns to the caller. Other than potential disk I/O waits, the functions are non-blocking and return to the caller as soon as they are finished.

libfs provides the caller with read and write modes of operation. In the read mode *libfs* provides functions to search the Frame Store on different attributes of frames and a function to retrieve a frame from the Frame Store. The Frame Store allows searching for frames based on the station sequence number (SSN), the data time of the frame, or the insert time (in this Frame Store) of the frame. A typical read application interacts with *libfs* using the following model:

1. Open the Frame Store.
2. Open the frame sets of interest.
3. Determine the time of the last Frame Store poll (by external means).
4. Poll for frame handles of frames arriving since this last poll time; update the poll time.
5. Using the frame handles, read the frames from the Frame Store.
6. Provide application processing of these frames.
7. Repeat steps 4–6 until the termination condition is met.
8. Close the frame sets.
9. Close the Frame Store.

A typical write-mode application interacts with *libfs* using the following model:

1. Open the Frame Store.
2. Open the frame sets of interest.
3. Create or receive by external means a new CD-1.1 frame.
4. Request a frame handle from *libfs*.

5. Store the frame using the frame handle.
6. Repeat steps 3–5 until the termination condition is met.
7. Close the frame sets.
8. Close the Frame Store.

libfs produces the following outputs:

- A return value for each function call. (See the *libfs(3)* man page for return values and error codes.)
- The error string, if set.
- Frame Store files (*.fs*) if operating in the mode of a Frame Store writer.
- Index files (*.flog*) if operating in the mode of a Frame Store writer.

Control

libfs is a statically linked library. Because *libfs* consists only of procedural functions, the library does not impose any control flow dependencies upon the calling application.

Interfaces

libfs manages the following two types of files. All access to the files is via *libfs*.

- Frame Store files (*.fs*)

Frame Store files are created by *libfs* as needed and contain CD-1.1 frames for a data time interval. Multiple files cover consecutive time intervals. When reading or writing a Data Frame, *libfs* uses the data time to determine the file containing the frame. Within a file the frames are unordered, hence the inclusion of a table of contents in each file. The Frame Store files are self contained: the table of contents lists the frame set name, time interval, number and identity of frames within, and so forth. These files may be archived without the need for an external index. The Frame Store typically contains many *.fs* files for each frame set.

▼ Detailed Design

Each Frame Store file is comprised of a table of contents followed by CD-1.1 frames. Incoming CD-1.1 frames are appended to the file until the initial table of contents is filled. At that point, another table of contents section is appended to the end, and the CD-1.1 frames may again be appended after this auxiliary table of contents.

Frame Store files are named *epoch.fs*, where *epoch* is an integer value returned by the UNIX *time(2)* call, for example, *972775800.fs*. *libfs* generates the names when a new file is required.

■ Index files (*.flog*)

Index files are preallocated during installation and contain references to CD-1.1 frames in Frame Store files. Index files provide rapid and efficient access to the Frame Store for recently stored frames. For each frame stored in a Frame Store file *libfs* creates a frame audit entry. These entries are ordered by insertion time. The Frame Store reading application polls this index file to discover recent additions to The Frame Store. The Frame Store contains one index file for each frame set.

Each index file is comprised of a header followed by frame audit entries. An audit entry contains the frame type, insertion time, data time, and several other parameters describing the frame. The audit entries occupy a circular buffer within the file; this permits new entries to overwrite old entries. After the audit entry is purged; access to those frames is still available by direct interrogation of the Frame Store files (via the table of contents).

Index files have names chosen via the configuration par-table. By convention, index files are named for the frame set, plus the frame log extension *.flog*, for example *UKR.flog*.

An additional file interface to *libfs* is the Frame Store configuration parameter file. This file is created and maintained outside of *libfs*. The library reads this file during the Frame Store open function to obtain configuration information. This information includes directory names, index filenames, file sizes, and so forth.

Error States

The library returns error codes when errors are discovered during processing. Some error codes have error messages associated with them; the library provides another function to return the error messages. The library returns an error code as soon as an error is discovered. When an error is discovered, any resources allocated during that function call are released prior to return.

Because *libfs* is dependent on file system objects, the most common form of error is a missing or incomplete file (either the `.fs` or `.flog` files). Other common errors are configuration errors in the configuration file. In the former case, it may be possible for the application to proceed by ignoring the intended processing for that frame. In the latter case access to the Frame Store is unavailable to the application, and the most prudent course of action for the application is to log the problem and terminate.

libcdo

libcdo encapsulates processing for creating and extracting information from messages/frames of the CD-1.1 protocol. The library contains processes that create frame data structures. Frame data structures, or frames, are used in IPC between processes adhering to the CD-1.1 protocol. Frames are intended as “containers” for the transport of self-describing information across a network. *libcdo* is used by applications whose functionality requires the creation or interpretation of the CD-1.1 frames. These applications provide *libcdo* frame building routines with the data elements, which are included in the frame. The frame building routines order the data based on the CD-1.1 protocol. Because these applications can be on machines of different architectures (big endian versus little endian), *libcdo* contains functions to convert frames from their host byte order (HBO) to network byte order (NBO) before transport.

▼ Detailed Design

Another function of *libcdo* is to extract data from frames. *libcdo* routines extract information from frames to allow user applications to identify frame routing information, subchannel information, signature authentication, and frame component sizes. As with the frame creation routines, the data extraction routines handle the conversion from NBO back to HBO.

Finally, *libcdo* routines are used for communication verification. For each frame created, a checksum is generated and attached to the frame. This is used by the receiving application to verify that the frame was transmitted without errors. The receiving application uses a *libcdo* routine to validate the checksum and verify that the frame was transmitted with no errors.

Input/Processing/Output

libcdo is a procedural-based library. Control flow remains with the calling application. *libcdo* routines execute to completion then return to the caller. For this reason, the majority of inputs to *libcdo* are calling arguments to the various routines supplied by calling applications. An exception to this rule is signature authentication. Even though the actual signature authentication is handled by another library, processing of the signature can be enabled/disabled by setting a flag.

Several inputs to calling applications indirectly affect *libcdo* and are parametric inputs to the calling application; they control the amount of data that *libcdo* outputs to log files. Inputs include *libcdo*-specific log classes and the priority of each of these classes.

Control

Control of *libcdo* is entirely contained within the calling application. *libcdo* is active only during the execution of one of its exported functions.

A calling application interfaces with *libcdo* through function calls. Data created by these functions are made available to the calling applications through pointers to data structures or by the return value of the functions.

Error States

Because *libcdo* is controlled by the calling application there are limited *libcdo* error states. The two most probable failures are invalid input data and memory allocation. In both cases the actions taken by *libcdo* are identical. For each failure *libcdo* logs an error message to the log file (provided that *libcdo* logging is enabled). After the logging is complete, the function returns and provides an error code to the calling application. The decision about how to proceed is left to the calling application.

MULTICAST PROTOCOL

Data Packet Format

[Table 4](#) shows the format of a multicast data packet. The data packet contains the frame set name, frame set number, frame sequence number, frame length, data packet number, frame fragment number, total fragments in the frame, and the frame fragment. The fragment number identifies which packet this is out of the total fragments in the frame (e.g. 1 out of 6). The total packet size is configurable but must be less than the smallest Maximum Transmission Unit (MTU) used by the transport network.

TABLE 4: FORMAT OF DATA PACKET

| Field | Format | Description |
|-----------------------|---------------|--|
| frame set name | 20-byte ASCII | name of the frame set |
| frame sequence number | IEEE integer | 4 least significant bytes of the frame sequence number assigned by the frame creator |
| frame length | IEEE integer | length of the frame |
| packet number | IEEE integer | data packet number assigned by the packet creator |
| fragment number | IEEE short | frame fragment number |
| total fragments | IEEE short | total number of fragments in the frame |
| frame fragment | variable | frame fragment |

PNack Packet Format

[Table 5](#) shows the format of a PNack packet. It contains the frame set name, the number of resend requests in the PNack packet, and a list of missing packet numbers. The number of data packet resend requests in a single PNack packet is limited to 100.

TABLE 5: FORMAT OF PNACK PACKET

| Field | Format | Description |
|--------------------|---------------|--|
| frameset name | 20-byte ASCII | name of the frame set |
| number of requests | IEEE integer | number of data packet resend requests |
| packet number 1 | IEEE integer | number of first data packet to resend |
| packet number 2 | IEEE integer | number of second data packet to resend |
| ... | | |
| packet number n | IEEE integer | number of nth data packet to resend |

DATABASE DESCRIPTION

The *CDS CD-1.1* depends on the database while establishing a connection for transmitting CD-1.1 frames and as the repository for data converted to CSS 3.0 format. These two uses are distinct and need not be concurrent. The first use is transient; after a connection is established the database is not needed for the remainder of the frame transmission session. The second use is continuous for a data center that needs rapid access to CSS 3.0 format data.

Database Interface

All *CDS CD-1.1* interaction with the database is through the generic database interface library, *libgdi*. *libgdi* is a dynamically loadable library that provides a simplified programmatic interface to the database.

Database Design

The entity-relationship model of the schema used by CDS CD-1.1 is shown in [Figure 40](#). *CDS CD-1.1* uses the **alphasite** and **dlman** tables for establishing connections. The **alphasite** table contains the names and addresses (IP) of valid data providers and is first checked when a connection attempt is made from a sending site. By using database row locking, the *CDS CD-1.1* prevents multiple concurrent connection attempts from the same data provider. The **dlman** table is consulted to route the incoming connection to the desired host machine and server process.

The *CDS CD-1.1* also uses the database for storing unpacked data segments in the CSS 3.0 data format. The *CDS CD-1.1* uses the **affiliation**, **instrument**, **sensor**, **site**, **sitechan**, and **wfconv** tables to read configuration information, and the **dlfile** and **wfdisc** tables to write descriptions of the new data. The input tables are read as needed. Several of the input tables are accessed through a view named **wfproto**. The **wfproto** view links to columns from the tables **instrument**, **site**, **sitechan**, and **wfconv**. The output table **dlfile** describes a disk loop file. A disk loop file may contain one or more segments of data, each segment of data is represented by a **wfdisc** record.

▼ Detailed Design

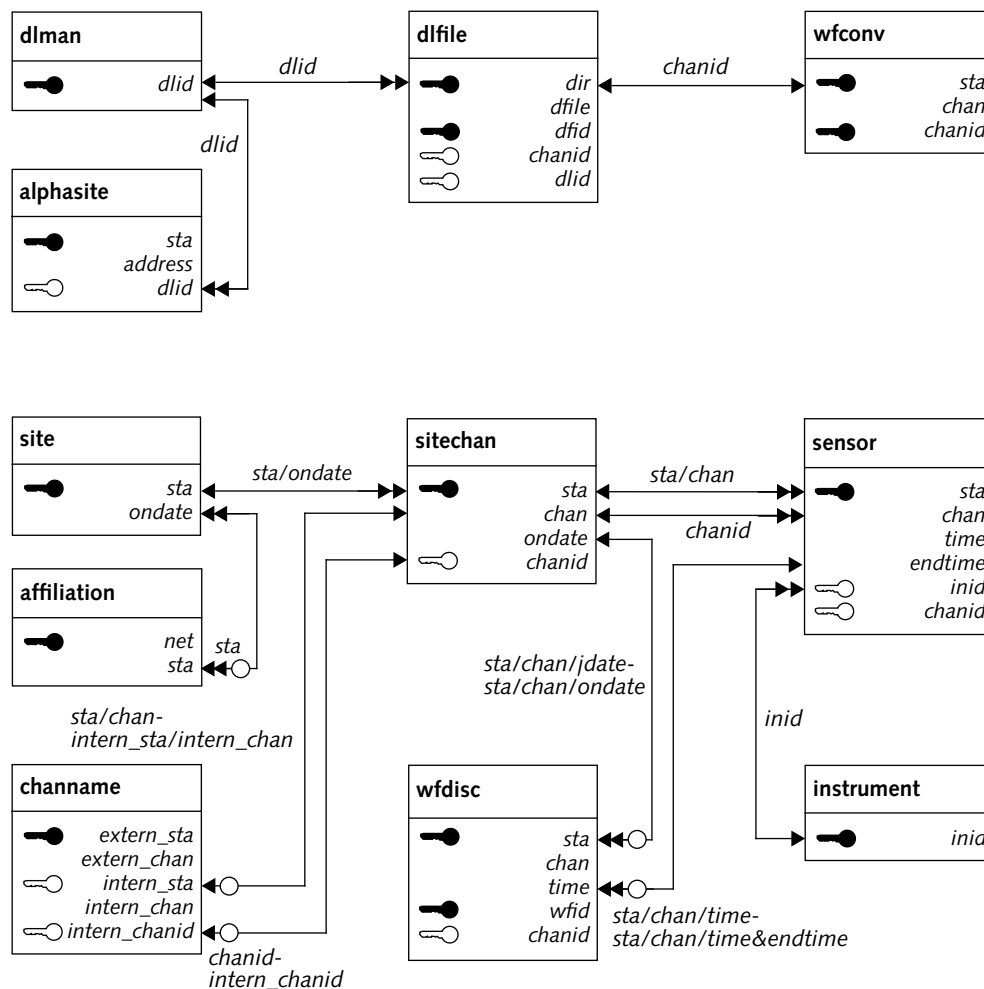


FIGURE 40. CDS CD-1.1 TABLE RELATIONSHIPS

CDS CD-1.1 use of the database tables is listed in [Table 6](#). For each table, the third column shows the purpose for reading or writing each attribute.

TABLE 6: DETAILED DATABASE USAGE BY CDS CD-1.1

| Table | Action | Usage of Attribute |
|--------------------|--------------|---|
| affiliation | reads | • <i>net</i> for record identification |
| | reads | • <i>sta</i> for identifying auxiliary network or station elements for creating station intervals |
| alphasite | reads | • <i>address</i> for verifying the connection address |
| | reads | • <i>prefdlid</i> for selecting the connection host ID |
| dlfile | writes | • <i>dir</i> for identifying disk loop directory |
| | writes | • <i>dfile</i> for identifying disk loop file |
| | reads/writes | • <i>machine</i> for identifying (data parsing) execution host |
| | writes | • <i>dfid</i> for specifying the numeric identifier of disk loop files |
| | writes | • <i>full</i> and <i>archive</i> for identifying status |
| | reads/writes | • <i>time</i> for identifying time of wfdisc record creation |
| | reads/writes | • <i>reaptime</i> for identifying time of <i>Data Parser</i> termination |
| | reads/writes | • <i>sta</i> for identifying provider station |
| | reads/writes | • <i>chan</i> and <i>chanid</i> for identifying channel |
| | write | • <i>dlid</i> for identifying writing <i>Data Parser</i> |
| dlman | reads | • <i>dlid</i> for identifying connection host (correlates with <i>prefdlid</i> in alphasite table) |
| | reads | • <i>machine</i> for identifying host computer |
| | reads/writes | • <i>running</i> signifies if data parsing is executing on specified host |
| instrument | reads | • <i>inid</i> for specifying the numeric identifier of instrument |
| | reads | • <i>instype</i> for identifying instrument type |
| | reads | • <i>ncalib</i> for specifying the calibration value for instrument |
| | reads | • <i>ncapler</i> for specifying the calibration period value |
| site | reads | • <i>sta</i> and <i>ondate</i> for record identification |
| | reads | • <i>statype</i> to distinguish between 3-C and array seismic station types |
| | reads | • <i>site</i> record for location library |

▼ Detailed Design

TABLE 6: DETAILED DATABASE USAGE BY CDS CD-1.1 (CONTINUED)

| Table | Action | Usage of Attribute |
|----------|--------|---|
| sitechan | reads | • <i>sta</i> for identifying data provider |
| | reads | • <i>chan</i> for identifying sensor channel |
| wfconv | reads | • <i>sta</i> for identifying data provider |
| | reads | • <i>chan</i> for identifying sensor channel |
| | reads | • <i>chanid</i> for specifying the numeric identifier of sensor channel |
| | reads | • <i>inauth</i> for identifying authentication of input |
| | reads | • <i>incomp</i> for identifying use of data compression |
| | reads | • <i>intype</i> for identifying input data type |
| | reads | • <i>insamp</i> for specifying input sample rate |
| | reads | • <i>outtype</i> for specifying output data type |
| | reads | • <i>outsamp</i> for specifying output <i>samp</i> rate (must currently equal input <i>samp</i> rate) |
| wfdisc | writes | • <i>sta</i> for identifying data provider |
| | writes | • <i>chan</i> for identifying sensor channel |
| | writes | • <i>time</i> for specifying start time of data |
| | writes | • <i>wfid</i> for identifying record number |
| | writes | • <i>chanid</i> for specifying the numeric identifier of sensor |
| | writes | • <i>jdate</i> for specifying data of data |
| | writes | • <i>endtime</i> for specifying end time of data |
| | writes | • <i>nsamp</i> for specifying the number of data samples in record |
| | writes | • <i>samprate</i> for specifying data sample rate |
| | writes | • <i>calib</i> for identifying calibration value |
| | writes | • <i>calper</i> for identifying current calibration period value |
| | writes | • <i>instype</i> for specifying instrument/sensor type |
| | writes | • <i>segtype</i> for specifying segment type (currently a constant assignment, "o") |
| | writes | • <i>datatype</i> for identifying output data type |
| | writes | • <i>dir</i> for identifying disk loop directory |
| | writes | • <i>dfile</i> for identifying disk loop file |
| | writes | • <i>foff</i> for specifying data offset into disk loop file |

Chapter 4: Requirements

This chapter describes the requirements of *CDS CD-1.1* and includes the following topics:

- [Introduction](#)
- [Functional Requirements](#)
- [System Requirements](#)
- [Requirements Traceability](#)

Chapter 4: Requirements

INTRODUCTION

The requirements of *CDS CD-1.1* can be categorized as general, functional, or system requirements. General requirements are nonfunctional aspects of *CDS CD-1.1*. These requirements express goals, design objectives, and similar constraints that are qualitative properties of the software. The degree to which these requirements are actually met can only be judged qualitatively. Functional requirements describe what *CDS CD-1.1* is to do and how it is to do it. System requirements pertain to general constraints, such as compatibility with other IDC subsystems, use of recognized standards for formats and protocols, and incorporation of standard subprogram libraries.

FUNCTIONAL REQUIREMENTS

The requirements described in this section are categorized by function.

Connection Manager Requirements

Connection Manager establishes validated connections between protocol peers. *Connection Manager* maintains the addressing information and connection policy of *CDS CD-1.1*. In traditional unicast and unicast catchup operation, after the connections are established, *Frame Exchange* communicates over the established private link without any further assistance from *Connection Manager*. In multicast operation, at a station data provider, delivery of data is performed by *Multicast Sender* on separate multicast connections. *Multicast Sender* makes no use of the private link established by *Connection Manager*.

Connection Manager has the following requirements:

1. *Connection Manager* shall support connection establishment protocol defined in [\[IDC3.4.3Rev0.2\]](#).
2. *Connection Manager* shall accept Connection Response Frame frames on a well-known port.
3. *Connection Manager* shall verify the authenticity of the requester, including the number of active connections for this requester and not respond to invalid requests.
4. *Connection Manager* shall invoke a *Frame Exchange* to service valid connections.
5. *Connection Manager* shall maintain connection status in a database table.
6. *Connection Manager* shall be highly available (*inetd* or daemon with monitors).
7. Upon boot *Connection Manager* shall be input driven.
8. *Connection Manager* shall have the ability to distribute *Frame Exchange* instances on the LAN.

Data Center Manager Requirements

Data Center Manager provides control processing for other CDS CD-1.1 processing entities that are to execute at a data center. This process control is provided for those processes that require monitoring and stimulation/instantiation from within the data center. This is in contrast to other processes that are instantiated by stimulation from outside of the data center. An example of an externally stimulated process is the process that services inbound data from an IMS station. An example of an internally instantiated process is *Data Parser*. *Data Center Manager* instantiates child processes at a data center. The child processes are monitored by *Data Center Manager* to detect exit conditions and configured output. When a child process exits, *Data Center Manager* has the capability to react to the condition via rules defined explicitly for the child. Reaction can include restarting the same process, starting a different process, or a null reaction. If a child process provides configured output to *Data Center Manager*, the manager has the ability to treat the output as a command, write the output to a log file, or ignore the output.

▼ Requirements

Data Center Manager has the following requirements:

9. *Data Center Manager* shall provide the ability to spawn and monitor CDS CD-1.1 processes on a host processor.
10. At a minimum managed processes shall include:
 - *Exchange Controller/Frame Exchange* pair
 - *Data Parser*
11. *Data Center Manager* shall provide the ability to restart managed processes in the event of a graceful or ungraceful termination.
12. *Data Center Manager* shall monitor the output of managed processes for error conditions. Monitoring may result in the sending of email, process termination, or the issuing of a signal to managed processes.
13. *Data Center Manager* shall provide the ability to receive commands to initiate the execution or termination of a managed process.

Connection Originator Requirements

Connection Originator provides a connection agent for processing suites, which are providers of CD-1.1 protocol data. It is also responsible for establishing an IP connection and providing CD-1.1 protocol transactions that result in an agreement for the exchange of protocol frames. *Connection Originator's* frame traffic is limited to that required for establishing a communications link. After the link is established, any frame may be sent across the link, though most generally the link is used for time-series Data Frames.

In traditional unicast and unicast catchup operation, *Connection Originator* exec's an *Exchange Controller* after a communication path is established. *Exchange Controller* inherits data descriptors for the communication link and provides the processing for further exchanges of protocol frames.

In multicast operation, *Connection Originator* exec's a *Multicast Receiver*. *Multicast Receiver* inherits the IP address and port number of the multicast group and the PNAck IP address and port number.

Connection Originator has the following requirements:

14. *Connection Originator* shall accept run-time configuration parameters to specify:
 - connection destination
 - time-out values
 - protocol designations
15. UDP and TCP connections shall be supported by *Connection Originator*.
16. *Connection Originator* shall allow retries when a connection request fails.
17. Successful and unsuccessful connections shall be logged.
18. The sequence and definition of connection frames shall be as documented in [\[IDC3.4.3Rev0.2\]](#).
19. Simultaneous execution of multiple *Connection Originators* shall be possible for connections to multiple destinations.
20. *Connection Originator* shall provide communication link information to an *Exchange Controller* when a successful connection has occurred.

Exchange Controller Requirements

Exchange Controller implements the policy of how to act upon protocol frames in traditional unicast and unicast catchup operation. *Exchange Controller* performs simple distribution and prioritization based on frame source and frame type for both inbound and outbound frames. *Exchange Controller* encapsulates the ordering policy for outbound frames and provides notifications for inbound/received frames.

Exchange Controller has the following requirements:

21. *Exchange Controller* shall control one and only one *Frame Exchange*.
22. Upon boot *Exchange Controller* shall prioritize and submit all unacknowledged Data Frames in the Frame Store to *Frame Exchange*.
23. *Exchange Controller* shall in coordination with *Frame Exchange*, maintain a Frame Log to provide a history of each frame handled.

▼ Requirements

24. *Exchange Controller* shall poll the Frame Store for the presence of new frames.
25. *Exchange Controller* shall order Data Frames and submit a notification to *Frame Exchange*.
26. *Exchange Controller* shall accept a frame message from *Frame Exchange* for communicating frame and processing status.
27. *Exchange Controller* shall provide a frame message to *Frame Exchange* to communicate frame and processing actions.
28. *Exchange Controller* shall notify *CDS CD-1.1* processes about the presence of newly received frames including Command Request Frames.

Frame Exchange Requirements

Frame Exchange's primary function is to reliably transport frames from one Frame Store to another. Each *Frame Exchange* has a single associated *Exchange Controller*, which directs the operation of *Frame Exchange* using frame messages sent over a pipe. Each *Frame Exchange* is connected by a TCP/IP socket to a corresponding *Frame Exchange* at another node to which it sends and from which it receives frames.

Frame Exchange has the following requirements:

29. *Frame Exchange* shall maintain a priority-ordered queue of the handles of frames to be sent.
30. *Frame Exchange* shall send the highest priority frame in its queue to the corresponding *Frame Exchange* on its attached socket and repeat.
31. *Frame Exchange* shall receive messages from its *Exchange Controller* directing it to add new frame handles to its queue.
32. *Frame Exchange* shall receive and periodically generate and send AckNack Frames to the corresponding *Frame Exchange* on its socket.
33. The AckNack Frame shall indicate which frames are available or needed in the referenced frame set.

34. *Frame Exchange* shall receive frames over its attached socket from its corresponding *Frame Exchange*.
35. The reception of an AckNack Frame shall cause the sending of a frame message to the associated *Exchange Controller* describing any frames that have been newly acknowledged.
36. Any frames that have been sent, but not acknowledged via the AckNack message, shall be queued for resending.
37. All frames other than AckNack Frames shall be stored in the appropriate frame set (denoted by the Creator/Destination of the frame). *Exchange Controller* associated with *Frame Exchange* will be notified by a frame message of the reception of the new frame.
38. *Frame Exchange* shall determine that a time-out has occurred if no messages are received within a time-out interval set by a configuration parameter.
39. In the event of a time-out the associated *Exchange Controller* shall be notified by a frame message.

Data Parser Requirements

Data Parser converts CD-1.1 protocol time-series data to a format used for signal processing and analysis software at a data center. *Data Parser* polls the frame sets for newly arrived Data Frames, which are parsed to extract station, channel, duration, data time, and compression information. Time series data are converted to the CSS 3.0 format and written to disk files in disk loop structures. Placement of data in disk loops is influenced by the size of the data and the represented time (of the data). *Data Parser* places data in disk loops in chronological order, even when the data's arrival has occurred out of (time) order. Information about the converted time series data is provided to a DBMS. The DBMS information allows other processes to acquire information about the signal data that are available for processing.

Data Parser has the following requirements:

40. *Data Parser* shall process CDS CD-1.1 Data Frames.

▼ Requirements

41. *Data Parser* shall obtain *CDS CD-1.1* Data Frames from the Frame Store.
42. *Data Parser* shall recognize duplicate input frames and process only one copy of the duplicate frame.
43. *Data Parser* shall recognize overlapping input frames and process the first-to-arrive frame. Overlap refers to both time and channel.
44. *Data Parser* shall uncompress data compressed according to a valid *CDS CD-1.1* compression algorithm(s).
45. *Data Parser* shall log parse failures and indicate the cause of the failure.
46. *Data Parser* shall filter input channels based on authentication status.
47. *Data Parser* shall use the existing DBMS and disk loop structure for storing time-series data.
48. The following pertains to data storage in disk loops:
 - Data shall be stored in chronological order.
 - Data shall not be moved after written to a disk loop.
 - DBMS data references shall be updated within N seconds of the data being written to the disk loops, where N is configurable.
49. *Data Parser* shall preserve sample timing of time-series data to within 1/sample-rate seconds.
50. *Data Parser* shall process up to a maximum of 100 channels allocated among up to 25 stations.
51. *Data Parser* shall support processing at 3x real-time for the maximum channel configuration.

Frame Store Stager Requirements

The *Frame Store Stager* provides an interface between the *CDS CD-1.1* and the Archiving Subsystem. Using this interface enables archiving of the Frame Store durable store on offline storage media.

The following are the requirements for the *Frame Store Stager*:

52. *Frame Store Stager* shall provide an interface between the *CDS CD-1.1* and the Archiving Subsystem.
53. *Frame Store Stager* shall be a stand-alone application that can process the Frame Store files while not interfering with current active Frame Store transactions.
54. *Frame Store Stager* shall maintain the integrity of the Frame Store files.
55. *Frame Store Stager* shall only process Frame Store files from active frame sets.
56. At a minimum, the following parameters shall be user-defined at runtime:
 - active Frame Set list
 - archiving time window (earliest and latest times)
 - archiving directory
57. *Frame Store Stager* shall log errors encountered during processing.

Authentication Signing Requirements

Authentication Signing provides the capability to apply DSA to CD-1.1 frames. Each frame as well as Channel Subframe in the CD-1.1 protocol contains authentication data fields. When a frame is created, the creator may issue a request to this entity to set and fill authentication data fields for carrying the digital signature of the creator. The *Authentication Signing* capability supports the specification of a key identifier used in the generation of the signature. Given the key ID and the frame to be signed, the frame signature field is filled and authentication data values are set to represent the presence and size of authentication data. The application of an authentication signature does not result in the encryption of the frame's content. The content is "out in the open"; only the signature is encrypted. *Authentication Signing* supports the application of signatures to all variants of protocol frames. When a Data Frame is submitted for signing, one signature is applied to the entire frame; separate signatures for the Channel Subframes are not applied. Each Channel Subframe to receive a signature must be addressed independently.

▼ Requirements

Authentication Signing has the following requirements:

58. *Authentication Signing* shall provide the ability to digitally sign protocol frames according to the description in [\[IDC3.4.3Rev0.2\]](#).
59. Information describing applied digital signatures shall be provided in the protocol frame including the following:
 - signature present/not-present
 - length of signature (in 8-bit bytes)
60. *Authentication Signing* shall provide the ability to digitally sign Channel Subframes according to the description in [\[IDC3.4.3Rev0.2\]](#).
61. *Authentication Signing* shall allow for the identification of a signature key to be used for signing a given frame or subframe, where each key identifies a separate signature.
62. *Authentication Signing* shall provide the ability to support a minimum of 10 signature keys.
63. The design of *Authentication Signing* shall not preclude the use of a hardware solution for providing digital signatures.
64. An error detected in signing a frame shall be communicated to the requesting software.
65. If *Authentication Signing* is unable to provide a signature, the provided (valid) frame or subframe shall be returned to the requester.

Signature Authentication Requirements

Signature Authentication supports verifying the authenticity of CD-1.1 frames. Each frame in the CD-1.1 protocol may contain a digital signature in the frame trailer. In addition, Channel Subframes support application of a signature. *Signature Authentication* provides the processing to validate digital signatures, where these signatures are evaluated within the context of the accompanying frame or subframe.

Signature Authentication has the following requirements:

66. *Signature Authentication* shall authenticate CD-1.1 frames according to the definition of frames in [\[IDC3.4.3Rev0.2\]](#).
67. Channel Subframes containing signatures according to the definition in [\[IDC3.4.3Rev0.2\]](#) shall be authenticated.
68. The results of frame and Channel Subframe authentication shall be made available for further processing.
69. When a signature fails authentication the containing frame or Channel Subframe shall be unaltered.
70. *Signature Authentication* shall support the ability to access a minimum of 10 authentication keys.
71. The execution of *Signature Authentication* shall be controllable via the run-time configuration.

Frame Store Requirements

The Frame Store is a data sink that houses both CD-1.1 protocol frames and an index to the frames. The Frame Store is transactional with respect to writes; the conclusion of a write operation guarantees that the frame is saved on disk and is accessible. The Frame Store stays within a preallocated disk capacity and recycles the space allocated to the oldest frames to make room for new frames. The Frame Store allows multiple processes to simultaneously retrieve frames from the store. The Frame Store is implemented as a set of time-bin disk files with a library interface to manage the files. The library that implements the Frame Store is called *libfs*.

The Frame Store has the following requirements:

72. *libfs* shall provide transactional frame processing such that an unambiguous and accurate result is provided for a request to store a frame.
73. *libfs* shall support the ability to provide requested frames in the exact form in which they were submitted.
74. *libfs* shall support storage of all types of CD-1.1 frames.

▼ Requirements

75. *libfs* shall support insertion of frames into the Frame Store with time values between the temporal minimum time value and the present.
76. *libfs* shall provide querying of frames by frame source and data time, entry time, or unique frame ID (frame set/sequence number).
77. Random access for the frames within the Frame Store shall be provided through *libfs*, such that any given frame (currently in the store) can be retrieved.
78. Access to the Frame Store by *libfs* shall support multiple non-blocking reads of the Frame Store.
79. *libfs* shall provide status/result response information to Frame Store requests.
80. Frame Store storage management shall provide capacity for at least seven days of Data Frames from each data source.
81. The Frame Store shall be self maintaining with respect to the maximum amount of data stored, such that space used by expired frames is recaptured.
82. *libfs* shall provide a means to initialize and configure the Frame Store under program control.
83. *libfs* shall support an interface that allows archiving of the Frame Store. This interface can also provide controls to record when Frame Store data are ready for archiving.

libcdo Requirements

libcdo is a library for creating and extracting frames in the format described in [\[IDC3.4.3Rev0.2\]](#). When creating frames *libcdo* can align bytes as required by the specification and swap bytes as required for network transport. When extracting frames the library provides a set of data structures and function calls, which allow access to the data contained in the protocol frames.

libcdo has the following requirements:

84. *libcdo* shall comply with [\[IDC3.4.3Rev0.2\]](#) for the basis of frame construction and deconstruction.
85. *libcdo* shall provide facilities for the creation of frames in NBO for transport.
86. *libcdo* shall provide facilities for converting received frames into data structures in HBO.

SYSTEM REQUIREMENTS

The CD-1.1 protocol is the next generation of the protocol used to transport time-series data from IMS stations to the IDC and NDCs. The CD-1.1 protocol is an enhancement of CD-1.0. The requirements for *CDS CD-1.1* evolved from the requirements of the CD-1.0 protocol. The following list identifies new system (high-level) requirements of the CDS in support of the CD-1.1 protocol.

87. CDS components using CD-1.1 protocol shall comply with [\[IDC3.4.3Rev0.2\]](#).
88. CDS shall support automated connections between CDS components communicating via CD-1.1 protocol (providers and consumers of data).
89. Audit capability shall be provided such that history of a given frame can be discovered.
90. *CDS CD-1.1* software shall provide time-series data in CSS 3.0 format.
91. A Frame Store shall be used to support transport and recovery of frames.
92. Frames stored in a Frame Store shall be retrievable as an exact replica of the originally stored frame.
93. The structure of a *Frame Store* shall permit archiving and restoration, and the ability to extract frames from a restored Frame Store.
94. The protocol Frame Store shall provide a durable store for frames.
95. Protocol frames between two CDS protocol participants shall be delivered with a reliability of no less than 99.9 percent averaged over a period of five days.

▼ Requirements

- 95a. The CDS shall deliver and parse 99.99 percent (Goal: 100.00 percent) of data that are available at a data provider into the data repository of a data consumer under nominal conditions (assuming no hardware failures at data provider or consumer).
- 95b. The CDS shall deliver and parse data into disk loops and the Oracle database at the data consumer within 3 minutes of the time the data are first available at the data provider during nominal conditions (assuming no communication outages or hardware failures at data provider or consumer).
- 95c. The CDS shall deliver and parse 99.9 percent of data that are available at a data provider into the data repository of a data consumer for data consumer outages caused by infrastructure failures of less than 4 days.
- 95d. The CDS shall deliver and parse 99.9 percent of data that are available at a data provider into the data repository of a data consumer for data provider outages caused by infrastructure failures of less than 4 days.
- 95e. The CDS shall deliver and parse data into disk loops and the Oracle database within a time no greater than 110% of (outage duration / excess bandwidth capacity).
- 95f. The CDS shall automatically attempt to initiate a connection between the data provider and data consumer within 5 minutes after recovery from an outage caused by an infrastructure failure.
- 95g. The CDS shall automatically resume service within 5 minutes after recovery from an outage caused by an infrastructure failure. Resumption of service includes processing connection requests, exchanging CD-1.1 frames, and putting data into the data repository.
- 96. Delivery of CD-1.1 Data Frames shall coexist with CD-1.0 protocol data.
- 97. CDS shall support automated forwarding of CD-1.1 protocol frames from the IDC.
- 98. The design of CDS shall allow potential use of UDP multicast for data distribution.

99. A priority policy for transmitting protocol frames shall be provided by CDS. Policy is established by the data provider and includes at least FIFO and LIFO ordering policies.
100. A software component suite shall be assembled that supports the construction of a System capable of participating in a CD-1.1 protocol network, as either a data provider or data center.
101. CDS shall perform authentication processing on frames containing authentication signatures and shall store the results.
102. The CDS shall provide multicasting capability to deliver CD-1.1 frames using underlying IP router multicast capability.

Multicast Subsystem Requirements

- M-1. Multiple data providers shall be capable of providing multicast data over the transport network.
- M-2. Multicasting shall support all IMS stations in the Treaty (up to 321 stations) in multicasting data over a common transport network.
- M-3. Transmission rates for multicast data shall be configurable to mitigate network congestion.
- M-4. A multicast data provider shall support 20 data consumers in a single multicast group.
- M-5. A multicast data provider shall not be limited in its sending by the absence or presence of any specific multicast data consumer.
- M-6. Multicasting shall support increases and decreases in the size of the multicast group without the need to restart the sending activity of the data provider.
- M-7. The size of multicast data packets shall be configurable to support the smallest MTU (Maximum Transmission Unit) used by the transport network.

▼ Requirements

- M-8. A method for identifying, requesting, and resending missing multicast data packets of an active multicast session shall be provided, limited by configurable data buffer size.
- M-9. A multicast data provider shall transmit frames in the order in which they were written to its CD-1.1 frame set, with the exception of retransmissions.
- M-10. A multicast data provider shall begin delivery at the current time or a configurable lookback time less than 10 minutes prior to the current time to prevent multicast stream gaps over short data provider outages.
- M-11. A multicast data consumer shall be capable of entering or leaving a multicast group without negatively impacting other group members.
- M-12. Multicasting shall require less bandwidth on the data-provider communication link to service four multicast data consumers than required by CD-1.1 point-to-point communications from a data provider directly to two point-to-point data consumers.
- M-13. A multicast group shall correspond to one and only one frame set.

**Unicast Catchup Subsystem
Requirements**

- C-1. The CDS shall use sequence numbers to detect frames not received by a data consumer via multicasting.
- C-2. The CDS shall attempt to deliver to a data consumer frames not received via multicasting.

REQUIREMENTS TRACEABILITY

The following tables trace *CDS CD-1.1* requirements to components and describe how the requirements are fulfilled.

**TABLE 7: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
CONNECTION MANAGER**

| | Requirement | How Fulfilled |
|----------|--|--|
| <u>1</u> | <i>Connection Manager</i> shall support connection establishment protocol defined in [IDC3.4.3Rev0.2] . | <i>Connection Manager</i> expects and processes a sequence of frames as defined in [IDC3.4.3Rev0.2] . |
| <u>2</u> | <i>Connection Manager</i> shall accept Connection Response Frame frames on a well-known port. | <i>Connection Manager</i> monitors a configured port for Connection Response Frame input. |
| <u>3</u> | <i>Connection Manager</i> shall verify the authenticity of the requester, including the number of active connections for this requester and not respond to invalid requests. | <i>Connection Manager</i> verifies all connection requests. |
| <u>4</u> | <i>Connection Manager</i> shall invoke a <i>Frame Exchange</i> to service valid connections. | <i>Connection Manager</i> starts an <i>Exchange Controller/Frame Exchange</i> pair for a traditional unicast or unicast catchup validated connection. |
| <u>5</u> | <i>Connection Manager</i> shall maintain connection status in a database table. | <i>Connection Manager</i> monitors unicast connection status but does not maintain it. This requirement was considered unnecessary in this implementation. |
| <u>6</u> | <i>Connection Manager</i> shall be highly available (<i>inetd</i> or daemon with monitors). | <i>Connection Manager</i> is configurable to be started by the Internet daemon process <i>inetd</i> . |
| <u>7</u> | Upon boot <i>Connection Manager</i> shall be input driven. | <i>Connection Manager</i> processes a Connection Response Frame received as input from a connection requesting source. |
| <u>8</u> | <i>Connection Manager</i> shall have the ability to distribute Frame Exchange instances on the LAN. | <i>Connection Manager</i> contacts a configured <i>Connection Manager Server</i> process via a socket connection. <i>Connection Manager Server</i> processes may exist on a variety of computers on the LAN. |

▼ Requirements

**TABLE 8: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
DATA CENTER MANAGER**

| | Requirement | How Fulfilled |
|--------------------|---|--|
| 9 | <i>Data Center Manager</i> shall provide the ability to spawn and monitor <i>CDS CD-1.1</i> processes on a host processor. | <i>Data Center Manager</i> is configured with a par file that allows the specification of processes/jobs to start and monitor. <i>Data Center Manager</i> spawns the configured process and waits for their termination. |
| 10 | At a minimum managed processes shall include: – <i>Exchange Controller/Frame Exchange</i> pair – <i>Data Parser</i> | Managed processes are specified in the configuration par file. |
| 11 | <i>Data Center Manager</i> shall provide the ability to restart managed processes in the event of a graceful or ungraceful termination. | A signal is received by <i>Data Center Manager</i> when a spawned process terminates. Based on configuration values, the terminated process may be restarted/spawned. |
| 12 | <i>Data Center Manager</i> shall monitor the output of managed processes for error conditions. Monitoring may result in the sending of email, process termination, or the issuing of a signal to managed processes. | <i>Data Center Manager</i> reads the UNIX standard output stream of started processes and can be configured to act on particular output. |
| 13 | <i>Data Center Manager</i> shall provide the ability to receive commands to initiate the execution or termination of a managed process. | <i>Data Center Manager</i> can receive input from the standard output stream of started processes. It can also accept connections to a configured socket port. Inputs from these sources may result in activation or termination of processes. |

**TABLE 9: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
CONNECTION ORIGINATOR**

| | Requirement | How Fulfilled |
|--------------------|--|---|
| 14 | <i>Connection Originator</i> shall accept run-time configuration parameters to specify: <ul style="list-style-type: none"> – connection destination – time-out values – protocol designations | Run-time parameters are accepted via the <i>libpar</i> par file mechanism. |
| 15 | UDP and TCP connections shall be supported by <i>Connection Originator</i> . | TCP is completely supported. UDP may be identified by a configuration parameter, but is not fully supported. |
| 16 | <i>Connection Originator</i> shall allow retries when a connection request fails. | <i>Connection Originator</i> may be configured to retry a connection after a failed attempt. |
| 17 | Successful and unsuccessful connections shall be logged. | <i>Connection Originator</i> logs connection processing results to its log file. |
| 18 | The sequence and definition of connection frames shall be as documented in [IDC3.4.3Rev0.2] . | <i>Connection Originator</i> issues and processes frames for establishing a connection in accordance with [IDC3.4.3Rev0.2] . |
| 19 | Simultaneous execution of multiple <i>Connection Originators</i> shall be possible for connections to multiple destinations. | Each <i>Connection Originator</i> instance executes independently from any other instance. |
| 20 | <i>Connection Originator</i> shall provide communication link information to an <i>Exchange Controller</i> when a successful connection has occurred. | In traditional unicast and unicast catchup operation, <i>Connection Originator</i> provides a protocol peer open socket file descriptor to the spawned <i>Exchange Controller</i> . |

▼ Requirements

**TABLE 10: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
EXCHANGE CONTROLLER**

| | Requirement | How Fulfilled |
|---|--|---|
| | <u>21</u> <i>Exchange Controller shall control one and only one Frame Exchange.</i> | <i>Exchange Controller spawns and interacts with a single Frame Exchange process.</i> |
| I | <u>22</u> Upon boot <i>Exchange Controller</i> shall prioritize and submit all unacknowledged Data Frames in the Frame Store to <i>Frame Exchange</i> . | In unicast operation, <i>Exchange Controller</i> scans its log of unacknowledged frames at startup and issues a request to <i>Frame Exchange</i> to resend unacknowledged frames. |
| | <u>23</u> <i>Exchange Controller</i> shall in coordination with <i>Frame Exchange</i> , maintain a Frame Log to provide a history of each frame handled. | <i>Exchange Controller</i> writes to a frame log to identify those frames queued to <i>Frame Exchange</i> for sending. |
| I | <u>24</u> <i>Exchange Controller</i> shall poll the Frame Store for the presence of new frames. | In unicast operation, <i>Exchange Controller</i> detects new frames to be sent by polling the Frame Store. |
| | <u>25</u> <i>Exchange Controller</i> shall order Data Frames and submit a notification to <i>Frame Exchange</i> . | <i>Exchange Controller</i> assigns a priority value to frames to be sent based on configuration values and submits them to <i>Frame Exchange</i> . |
| | <u>26</u> <i>Exchange Controller</i> shall accept a frame message from <i>Frame Exchange</i> for communicating frame and processing status. | <i>Exchange Controller</i> monitors its open communication pipe to <i>Frame Exchange</i> . When a frame message providing an acknowledgement is received, <i>Exchange Controller</i> updates its transaction log. |
| | <u>27</u> <i>Exchange Controller</i> shall provide a frame message to <i>Frame Exchange</i> to communicate frame and processing actions. | <i>Exchange Controller</i> provides a frame message through its communication pipe to <i>Frame Exchange</i> to request frames be sent and to communicate other control actions as necessary. |
| | <u>28</u> <i>Exchange Controller</i> shall notify CDS CD-1.1 processes about the presence of newly received frames including Command Request Frames. | <i>Exchange Controller</i> recognizes the receipt of different frame types. |

**TABLE 11: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
FRAME EXCHANGE**

| | Requirement | How Fulfilled |
|--------------------|--|---|
| 29 | <i>Frame Exchange</i> shall maintain a priority-ordered queue of the handles of frames to be sent. | <i>Frame Exchange</i> maintains a sending queue for frames to be sent. Frames from <i>Exchange Controller</i> are ordered in the queue according to their assigned priority. |
| 30 | <i>Frame Exchange</i> shall send the highest priority frame in its queue to the corresponding <i>Frame Exchange</i> on its attached socket and repeat. | <i>Frame Exchange</i> sends the highest priority frame from its send queue(s) first. |
| 31 | <i>Frame Exchange</i> shall receive messages from its <i>Exchange Controller</i> directing it to add new frame handles to its queue. | <i>Frame Exchange</i> polls the <i>Exchange Controller</i> for input. Input can identify new frames to be sent. |
| 32 | <i>Frame Exchange</i> shall receive and periodically generate and send AckNack Frames to the corresponding <i>Frame Exchange</i> on its socket. | <i>Frame Exchange</i> periodically produces AckNack Frames according to the run-time configuration. |
| 33 | The AckNack Frame shall indicate which frames are available or needed in the referenced frame set. | <i>Frame Exchange</i> interprets the AckNack Frame to identify gaps in sequence numbers. Each gap represents one or more frames that must be sent. |
| 34 | <i>Frame Exchange</i> shall receive frames over its attached socket from its corresponding <i>Frame Exchange</i> . | <i>Frame Exchange</i> reads from an open socket file descriptor connected to a peer <i>Frame Exchange</i> . Input received on this socket is interpreted as CD-1.1 frames. |
| 35 | The reception of an AckNack Frame shall cause the sending of a frame message to the associated <i>Exchange Controller</i> describing any frames that have been newly acknowledged. | <i>Frame Exchange</i> queues frame messages to <i>Exchange Controller</i> for each frame that is acknowledged. Frame messages are sent over the open pipe to <i>Exchange Controller</i> . |

▼ Requirements

TABLE 11: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
FRAME EXCHANGE (CONTINUED)

| Requirement | How Fulfilled |
|---|---|
| 36 Any frames that have been sent, but not acknowledged via the AckNack message, shall be queued for resending. | <i>Frame Exchange</i> maintains a list of those frames sent but not yet acknowledged. After a configurable time-out period, unacknowledged frames are resent. |
| 37 All frames other than AckNack Frames shall be stored in the appropriate frame set (denoted by the Creator/Destination of the frame). <i>Exchange Controller</i> associated with <i>Frame Exchange</i> will be notified by a frame message of the reception of the new frame. | With the exception of the AckNack Frame, <i>Frame Exchange</i> writes all frames received from its peer <i>Frame Exchange</i> to the appropriate frame set in the Frame Store. |
| 38 <i>Frame Exchange</i> shall determine that a time-out has occurred if no messages are received within a time-out interval set by a configuration parameter. | A time-out counter increments for each <i>Heartbeat</i> interval in which no AckNack message is received and resets to zero at the arrival of an AckNack. If the time-out counter reaches its configured maximum, a time-out condition is declared. |
| 39 In the event of a time-out the associated <i>Exchange Controller</i> shall be notified by a frame message. | <i>Frame Exchange</i> provides a time-out frame message to <i>Exchange Controller</i> whenever a time-out condition is declared. |

**TABLE 12: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
DATA PARSER**

| | Requirement | How Fulfilled |
|--------------------|---|--|
| 40 | <i>Data Parser shall process CDS CD-1.1 Data Frames.</i> | <i>Data Parser parses CD-1.1 Data Frames. Nonconforming frames are discarded.</i> |
| 41 | <i>Data Parser shall obtain CDS CD-1.1 Data Frames from the Frame Store.</i> | <i>Data Parser polls the Frame Store for new frames to be parsed and retrieves those frames.</i> |
| 42 | <i>Data Parser shall recognize duplicate input frames and process only one copy of the duplicate frame.</i> | <i>Data Parser recognizes duplicate input frames based on sequence number and discards duplicate frames.</i> |
| 43 | <i>Data Parser shall recognize overlapping input frames and process the first-to-arrive frame. Overlap refers to both time and channel.</i> | <i>Data Parser uses the database to record parsed data, which enables detection of overlapping input frames. If this requirement is not met, data are written for the last processed frame.</i> |
| 44 | <i>Data Parser shall uncompress data compressed according to a valid CDS CD-1.1 compression algorithm(s).</i> | <i>Data Parser supports decompression of data compressed with the Canadian Compression algorithm.</i> |
| 45 | <i>Data Parser shall log parse failures and indicate the cause of the failure.</i> | <i>Data Parser logs failed Channel Sub-frame processing information to a log file.</i> |
| 46 | <i>Data Parser shall filter input channels based on authentication status.</i> | <i>Data Parser filters input based on configuration parameters and the availability and validity of data, not on authentication status. This requirement was not considered to be correct as stated.</i> |
| 47 | <i>Data Parser shall use the existing DBMS and disk loop structure for storing time-series data.</i> | <i>Data Parser writes parsed data in CSS 3.0 format and constructs wfdisc records for the written data.</i> |

▼ Requirements

**TABLE 12: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
DATA PARSER (CONTINUED)**

| | Requirement | How Fulfilled |
|--------------------|--|--|
| 48 | <p>The following pertains to data storage in disk loops:</p> <ul style="list-style-type: none"> – Data shall be stored in chronological order. – Data shall not be moved after written to a disk loop. – DBMS data references shall be updated within N seconds of the data being written to the disk loops, where N is configurable. | <i>Data Parser</i> writes data to disk loops and wfdisc records in an ordered, deterministic method. |
| 49 | <i>Data Parser</i> shall preserve sample timing of time-series data to within 1/sample-rate seconds. | <i>Data Parser</i> preserves sample time received in Channel Subframes. |
| 50 | <i>Data Parser</i> shall process up to a maximum of 100 channels allocated among up to 25 stations. | <i>Data Parser</i> processing is configured with par file parameters that can accommodate identification of 100 channels and 25 stations. |
| 51 | <i>Data Parser</i> shall support processing at 3x real-time for the maximum channel configuration. | Parsing of data by <i>Data Parser</i> is an efficient operation and taken singly satisfies this requirement. However, data parsing also includes authentication of Channel Subframes, which is computationally expensive. This requirement may or may not be satisfied based on the hardware of the host platform when authentication is considered. |

**TABLE 13: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
FRAME STORE STAGER**

| Requirement | How Fulfilled |
|---|---|
| 52 <i>Frame Store Stager</i> shall provide an interface between the <i>CDS CD-1.1</i> and the Archiving Subsystem. | <i>Frame Store Stager</i> interfaces with the Frame Store via the UNIX file system, and with the Archiving Subsystem by using Archiving's interface methodology of fileproduct records in the DBMS. |
| 53 <i>Frame Store Stager</i> shall be a stand-alone application that can process the Frame Store files while not interfering with current active Frame Store transactions. | <i>Frame Store Stager</i> is constructed as a stand-alone executable. It retrieves data from the Frame Store using non-blocking read-only access. |
| 54 <i>Frame Store Stager</i> shall maintain the integrity of the Frame Store files. | Frame Store files are moved from the Frame Store to a staging area. <i>Frame Store Stager</i> maintains a list of previously moved Frame Store files to ensure that late arriving data do not cause the original Frame Store file to be overwritten. |
| 55 <i>Frame Store Stager</i> shall only process Frame Store files from active frame sets. | Configuration data are provided to <i>Frame Store Stager</i> to identify frame sets to be archived. Further, <i>Frame Store Stager</i> only archives files that are updated during the archive window as specified via configuration parameters. |
| 56 At a minimum, the following parameters shall be user-defined at runtime: <ul style="list-style-type: none"> – active Frame Set list – archiving time window (earliest and latest times) – archiving directory | <i>Frame Store Stager</i> uses <i>libpar</i> to read configuration parameters for frame sets, archiving time window, and staging directory(ies). |
| 57 <i>Frame Store Stager</i> shall log errors encountered during processing. | <i>Frame Store Stager</i> logs all significant processing events and error conditions using the <i>liblog</i> library. |

▼ Requirements

**TABLE 14: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
AUTHENTICATION SIGNING**

| Requirement | How Fulfilled |
|--|--|
| 58 <i>Authentication Signing</i> shall provide the ability to digitally sign protocol frames according to the description in [IDC3.4.3Rev0.2] . | Digital signatures are applied to created frames according to definitions in [IDC3.4.3Rev0.2] . |
| 59 Information describing digital signatures shall be provided in the protocol frame including the following: – signature present/not-present – length of signature (in 8-bit bytes) | Signed frames have authentication fields in the protocol frame filled in with descriptive data. |
| 60 <i>Authentication Signing</i> shall provide the ability to digitally sign Channel Subframes according to the description in [IDC3.4.3Rev0.2] . | Digital signatures are applied using <i>libcdo</i> and <i>libas</i> to create Channel Subframes according to the definitions in [IDC3.4.3Rev0.2] . However, Channel Subframes are not signed in the data center configuration of <i>CDS CD-1.1</i> . |
| 61 <i>Authentication Signing</i> shall allow for the identification of a signature key to be used for signing a given frame or subframe, where each key identifies a separate signature. | Configuration parameters are used by application software to identify the signature key to use in signing created frames. |
| 62 <i>Authentication Signing</i> shall provide the ability to support a minimum of 10 signature keys. | An index file is used to provide a database of signature keys available to a signing application and supports an arbitrary number of signature keys. |

**TABLE 14: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
AUTHENTICATION SIGNING (CONTINUED)**

| | Requirement | How Fulfilled |
|--------------------|---|--|
| 63 | The design of <i>Authentication Signing</i> shall not preclude the use of a hardware solution for providing digital signatures. | The design of <i>libas</i> allows enhancement to use an API to a hardware signing solution. |
| 64 | An error detected in signing a frame shall be communicated to the requesting software. | Return data from signing software provides a success or failure result status. |
| 65 | If <i>Authentication Signing</i> is unable to provide a signature, the provided (valid) frame or subframe shall be returned to the requester. | <i>libas</i> returns an unaltered frame to the requester if it is not able to apply a signature. |

**TABLE 15: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
SIGNATURE AUTHENTICATION**

| | Requirement | How Fulfilled |
|--------------------|--|--|
| 66 | <i>Signature Authentication</i> shall authenticate CD-1.1 frames according to the definition of frames in [IDC3.4.3Rev0.2] . | Processes that receive/process frames request authentication through the API of <i>libas</i> . Digital signatures are authenticated according to definitions in [IDC3.4.3Rev0.2] . |
| 67 | Channel Subframes containing signatures according to the definition in [IDC3.4.3Rev0.2] shall be authenticated. | <i>Data Parser</i> examines Channel Subframes as they are being parsed and requests authentication processing when signed. |
| 68 | The results of frame and Channel Subframe authentication shall be made available for further processing. | Success or failure of authentication signatures is written to frame logs. |

▼ Requirements

**TABLE 15: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
SIGNATURE AUTHENTICATION (CONTINUED)**

| Requirement | How Fulfilled |
|---|---|
| 69 When a signature fails authentication the containing frame or Channel Sub-frame shall be unaltered. | Authentication processing is only concerned with the validity of a digital signature. If a signature does not validate, that result is returned to the requester. The frame or Channel Sub-frame is unaltered by the determination. |
| 70 <i>Signature Authentication</i> shall support the ability to access a minimum of 10 authentication keys. | An index file is used to provide a database of an arbitrary number of signature keys available to a verifying application. |
| 71 The execution of <i>Signature Authentication</i> shall be controllable via the run-time configuration. | A run-time configuration parameter is provided to the <i>libas</i> library to enable or disable verification processing. |

**TABLE 16: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
FRAME STORE**

| Requirement | How Fulfilled |
|---|---|
| 72 <i>libfs</i> shall provide transactional frame processing such that an unambiguous and accurate result is provided for a request to store a frame. | The logic of <i>libfs</i> methods is structured so that I/O for frames occurs before the update of "table of contents" type structures. Indices reflect actual store contents. Additionally, store operations are atomic from the perspective of the calling program. |
| 73 <i>libfs</i> shall support the ability to provide requested frames in the exact form in which they were submitted. | <i>libfs</i> supports search and retrieval of stored frames. Retrieved frames are exact images of stored frames. Searches are made by sequence number, data time, or log time. |
| 74 <i>libfs</i> shall support storage of all types of CD-1.1 frames. | Any CD-1.1 frame type may be submitted for storage by <i>libfs</i> methods. |

**TABLE 16: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
FRAME STORE (CONTINUED)**

| | Requirement | How Fulfilled |
|--------------------|---|--|
| 75 | <i>libfs</i> shall support insertion of frames into the Frame Store with time values between the temporal minimum time value and the present. | <i>libfs</i> stores frames in files that relate to a Frame Store time bin. A Data Frame is not stored only if its time is before the time range of "time now corner" and "time now corner" minus the duration of the Frame Store. |
| 76 | <i>libfs</i> shall provide querying of frames by frame source and data time, entry time, or unique frame ID (frame set/sequence number). | <i>libfs</i> allows searches by sequence number, data time, or log time within a given frame set. Frame sets represent specification of source. |
| 77 | Random access for the frames within the Frame Store shall be provided through <i>libfs</i> , such that any given frame (currently in the store) can be retrieved. | <i>libfs</i> uses <i>libframelog</i> to manage indices into the Frame Store files. Frame logs provide information that enables random access to frames in the Frame Store. |
| 78 | Access to the Frame Store by <i>libfs</i> shall support multiple non-blocking reads of the Frame Store. | Read-only access to the Frame Store is not restricted by the logic of <i>libfs</i> . |
| 79 | <i>libfs</i> shall provide status/result response information to Frame Store requests. | <i>libfs</i> methods provide a return value indicating the success or failure of requested operations. When a failure is reported most methods also provide an error number that can be used to retrieve additional information about the failure. |
| 80 | The Frame Store storage management shall provide capacity for at least seven days of Data Frames from each data source. | The Frame Store capacity is provided with configuration parameters. Processing can easily support a seven day duration. Available disk space is the limiting factor. |

▼ Requirements

**TABLE 16: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
FRAME STORE (CONTINUED)**

| Requirement | How Fulfilled |
|--|---|
| 81 The Frame Store shall be self maintaining with respect to the maximum amount of data stored, such that space used by expired frames is recaptured. | <i>libfs</i> maintains its own data files to eliminate/delete old files when the configured maximum duration is exceeded as new ones are created. |
| 82 <i>libfs</i> shall provide a means to initialize and configure the Frame Store under program control. | <i>libfs</i> supports providing control in an open request to create needed files if they do not already exist. Creations are based on the definitions in the Frame Store par file. |
| 83 <i>libfs</i> shall support an interface that allows archiving of the Frame Store. This interface can also provide controls to record when the Frame Store data are ready for archiving. | Frame Store files can be archived as any other data file. Because files are self describing, the information required to use the archived file is captured simultaneously. |

**TABLE 17: TRACEABILITY OF FUNCTIONAL REQUIREMENTS:
LIBCDO REQUIREMENTS**

| Requirement | How Fulfilled |
|---|--|
| 84 <i>libcdo</i> shall comply with IDC3.4.3Rev0.2 for the basis of frame construction and deconstruction. | <i>libcdo</i> has data definition structures that capture the format of each CD-1.1 frame. These structures are used to create and decompose frames in compliance with the formats and protocol specification. |
| 85 <i>libcdo</i> shall provide facilities for the creation of frames in NBO for transport. | Frames created with methods from <i>libcdo</i> are provided to the requester in NBO. |
| 86 <i>libcdo</i> shall provide facilities for converting received frames into data structures in HBO. | Frames submitted to <i>libcdo</i> for decomposition are expected to be in NBO. The result of decomposition is data structures in HBO. |

TABLE 18: TRACEABILITY OF SYSTEM REQUIREMENTS

| | Requirement | How Fulfilled |
|--------------------|---|--|
| 87 | CDS components using CD-1.1 protocol shall comply with [IDC3.4.3Rev0.2] . | All components of the <i>CDS CD-1.1</i> comply with descriptions provided in [IDC3.4.3Rev0.2] . |
| 88 | The CDS shall support automated connections between CDS components communicating via CD-1.1 protocol (providers and consumers of data). | The designs of <i>Connection Manager</i> and <i>Connection Originator</i> enable automated connection processing between protocol peers. |
| 89 | Audit capability shall be provided such that history of a given frame can be discovered. | <i>CDS CD-1.1</i> components use the Frame Store as a data store for frames. The Frame Store supports the discovery of a frame's history as it progresses through the system. |
| 90 | <i>CDS CD-1.1</i> software shall provide time-series data in CSS 3.0 format. | <i>CDS CD-1.1</i> components deliver CD-1.1 Data Frames containing time-series data. Time-series data are parsed from Data Frames into CSS 3.0 format by <i>Data Parser</i> . |
| 91 | A Frame Store shall be used to support transport and recovery of frames. | The Frame Store is used as a data store for frames. Frames to be sent are deposited and retrieved from a Frame Store. Frames received are deposited in a Frame Store. |
| 92 | Frames stored in a Frame Store shall be retrievable as an exact replica of the originally stored frame. | The Frame Store library, <i>libfs</i> , used by <i>CDS CD-1.1</i> applications supports search and retrieval of stored frames. Retrieved frames are exact images of stored frames. Searches are made by sequence number, data time, or log time. |
| 93 | The structure of the Frame Store shall permit archiving and restoration, and the ability to extract frames from a restored Frame Store. | The Frame Store is composed of self-describing files that are available for archiving. When a Frame Store file(s) is restored, the self-describing attribute of the file(s) enables retrieval/extraction of archived frames. |

▼ Requirements

TABLE 18: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

| | Requirement | How Fulfilled |
|---------------------|---|---|
| 94 | The protocol Frame Store shall provide a durable store for frames. | Frame store files are written to disk files. |
| 95 | Protocol frames between two CDS protocol participants shall be delivered with a reliability of no less than 99.9 percent averaged over a period of 5 days. | <p>This requirement is fulfilled by the design of all of the <i>CDS CD-1.1</i> application software and the interactions between them, in particular:</p> <p><i>Data Center Manager</i> monitors and restarts processes</p> <p><i>Connection Manager</i> prevents multiple connections by a single site</p> <p><i>Connection Manager Server</i> distributes connection servicing to multiple hosts on the data center LAN</p> <p><i>Exchange Controller</i> requests resends of frames not acknowledged in a previous execution</p> <p><i>Frame Exchange</i> resends frames until an acknowledgement is received</p> <p>Frame Store provides a durable store for frames sent and received</p> |
| 95a | The CDS shall deliver and parse 99.99 percent (Goal: 100.00 percent) of data that are available at a data provider into the data repository of a data consumer under nominal conditions (assuming no hardware failures at data provider or consumer). | |
| 95b | The CDS shall deliver and parse data into disk loops and the Oracle database at the data consumer within 3 minutes of the time the data are first available at the data provider during nominal conditions (assuming no communication outages or hardware failures at data provider or consumer). | |

TABLE 18: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

| | Requirement | How Fulfilled |
|--|---|---------------|
| | <u>95c</u> The CDS shall deliver and parse 99.9 percent of data that are available at a data provider into the data repository of a data consumer for data consumer outages caused by infrastructure failures of less than 4 days. | |
| | <u>95d</u> The CDS shall deliver and parse 99.9 percent of data that are available at a data provider into the data repository of a data consumer for data provider outages caused by infrastructure failures of less than 4 days. | |
| | <u>95e</u> The CDS shall deliver and parse data into disk loops and the Oracle database within a time no greater than 110% of (outage duration / excess bandwidth capacity). | |
| | <u>95f</u> The CDS shall automatically attempt to initiate a connection between the data provider and data consumer within 5 minutes after recovery from an outage caused by an infrastructure failure. | |
| | <u>95g</u> The CDS shall automatically resume service within 5 minutes after recovery from an outage caused by an infrastructure failure. Resumption of service includes processing connection requests, exchanging CD-1.1 frames, and putting data into the data repository. | |

▼ Requirements

TABLE 18: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

| Requirement | How Fulfilled |
|---|---|
| 96 Delivery of CD-1.1 Data Frames shall coexist with CD-1.0 protocol data. | The design of the <i>CDS CD-1.1</i> applications does not rely on any <i>CDS CD-1.0</i> applications. CD-1.1 data are parsed into disk loop files, which are the only common structures between the two software implementations. This sharing is in format only; <i>DLMAN</i> and <i>DLParse</i> do not write to the same files. |
| 97 CDS shall support automated forwarding CD-1.1 protocol frames from the IDC. | Configuration of a <i>Data Center Manager</i> , <i>Connection Originator</i> , <i>Exchange Controller</i> , and <i>Frame Exchange</i> provide automated frame forwarding capability. |
| 98 The design of CDS shall allow potential use of UDP multicast for data distribution. | The design of <i>CDS CD-1.1</i> includes UDP multicast frame transport. |
| 99 A priority policy for transmitting protocol frames shall be provided by the CDS. Policy is established by the data provider and includes at least FIFO and LIFO ordering policies. | <i>Exchange Controller</i> accepts ordering policies and priorities as configuration parameters. These parameters are used to construct a priority value to provide with a frame transmission request. <i>Frame Exchange</i> acts on the priority to send frames in the desired order/sequence. |
| 100 A software component suite shall be assembled that supports the construction of a System capable of participating in a CD-1.1 protocol network, as either a data provider or data center. | A public bundle of <i>CDS CD-1.1</i> software has been created and released. The bundle contains elements useful to a data provider implementer. The element of this requirement that identifies support for a data center is not satisfied. |
| 101 CDS shall perform authentication processing on frames containing authentication signatures and shall store the results. | All applications that create and receive frames are capable of creating/verifying authentication signatures. Results from authentication activity are, at a minimum, written to log files. Some processes also store authentication results in frame logs. |

TABLE 18: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

| | Requirement | How Fulfilled |
|---------------------|--|--|
| 102 | The CDS shall provide multicasting capability to deliver CD-1.1 frames using underlying IP router multicast capability. | <i>Connection Originator</i> and <i>Connection Manager</i> will be modified to support multicast connections. A separate multicast subsystem will perform multicast frame transport. The design for multicast operation is presented in many sections of this document. |
| M-1 | Multiple data providers shall be capable of providing multicast data over the transport network. | Different multicast data providers will be configured to use different multicast addresses. |
| M-2 | Multicasting shall support all IMS stations in the Treaty (up to 321 stations) in multicasting data over a common transport network. | Each station will use a different multicast data provider and different multicast address. |
| M-3 | Transmission rates for multicast data shall be configurable to mitigate network congestion. | The transmission rate is provided in a parameter file and used by the multicast data provider to control its sending rate. |
| M-4 | A multicast data provider shall support 20 data consumers in a single multicast group. | The mechanism that depends on group size is packet retransmission. The mechanism used is believed to scale to 20 receiver per group. |
| M-5 | A multicast data provider shall not be limited in its sending by the absence or presence of any specific multicast data consumer. | The initiation and continued operation of the multicast data provider is independent of multicast consumers. |
| M-6 | Multicasting shall support increases and decreases in the size of the multicast group without the need to restart the sending activity of the data provider. | Changes in network distribution of multicast packets are accomplished automatically by routers. The only impact on the multicast data provider of changes in group membership are the addition or deletion of a potential source of PNacks. This change doesn't require any special action on the part of the multicast data provider. |

▼ Requirements

TABLE 18: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

| Requirement | How Fulfilled |
|---|---|
| M-7 The size of multicast data packets shall be configurable to support the smallest MTU (Maximum Transmission Unit) used by the transport network. | The size of the multicast data packets is given in a parameter file and used by the multicast data provider. |
| M-8 A method for identifying, requesting, and resending missing multicast data packets of an active multicast session shall be provided, limited by configurable data buffer size. | This is provided by the PNack mechanism. |
| M-9 A multicast data provider shall transmit frames in the order in which they were written to its CD-1.1 frame set, with the exception of retransmissions. | The multicast data provider reads frames in FIFO order, inserts their data in the same order in its buffer, and sends the data in the same order. |
| M-10 A multicast data provider shall begin delivery at the current time or a configurable lookback time less than 10 minutes prior to the current time to prevent multicast stream gaps over short data provider outages. | The lookback interval is given in a parameter file and used by the multicast data provider to decide where to start reading the frame store. |
| M-11 A multicast data consumer shall be capable of entering or leaving a multicast group without negatively impacting other group members. | There is no direct interaction among the multicast consumers that belong to the same group. |
| M-12 Multicasting shall require less bandwidth on the data-provider communication link to service four multicast data consumers than required by CD-1.1 point-to-point communications from a data provider directly to two point-to-point data consumers. | The design is believed to provide this level of efficiency. Verification that the level has been achieved will be by actual measurement. |

TABLE 18: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

| | Requirement | How Fulfilled |
|--|---|--|
| | M-13 A multicast group shall correspond to one and only one frame set. | A multicast provider is configured with a single frame set. It only reads from that frame set. |
| | C-1 The CDS shall use sequence numbers to detect frames not received by a data consumer via multicasting. | This functionality is provide by <i>Missing Frame Detector</i> . |
| | C-2 The CDS shall attempt to deliver to a data consumer frames not received via multicasting. | This functionality is provided by the unicast subsystem when it operates in catchup mode. |

References

The following sources supplement, or are referenced in, the document:

- | | |
|------------------|---|
| [Aga01] | Agarwal, D., "Discussion of Reliable Multicast Progress for the Continuous Data Protocol," <i>GCI Workshop, Vienna, Austria, October 1-3, 2001</i> . |
| [DOD94a] | Department of Defense Data Item Description, <i>Software Design Description</i> , DI-IPSC-81435, 1994. |
| [DOD94b] | Department of Defense Data Item Description, <i>Software Requirements Specification</i> , DI-IPSC-81433, 1994. |
| [IDC3.4.1Rev3] | Science Applications International Corporation, Veridian Systems, <i>Formats and Protocols for Messages, Revision 3</i> , SAIC-01/3053, TN-2865, 2001. |
| [IDC3.4.2Rev0.1] | Science Applications International Corporation, <i>Formats and Protocols for Continuous Data CD-1.0, Revision 0.1</i> , SAIC-01/3054, 2002. |
| [IDC3.4.3Rev0.2] | Science Applications International Corporation, <i>Formats and Protocols for Continuous Data CD-1.1, Revision 0.2</i> , SAIC-01/3027Rev0.2, 2001. |
| [IDC5.1.1Rev3] | Science Applications International Corporation, Veridian Systems, <i>Database Schema, (Part 1, Part 2, and Part 3), Revision 3</i> , SAIC-01/3052, TN-2866, 2001. |
| [IDC6.5.18] | Science Applications International Corporation, <i>Continuous Data Subsystem CD-1.1 Software User Manual</i> , SAIC-01/3006, 2001. |

▼ References

- [SAIC-01/3068] Science Applications International Corporation, *Options for Reliable Multicast Transport of CD-1.1 Frames*, SAIC-01/3068, 2001.

Glossary

A

AckNack

Acknowledgement/Negative Acknowledgement.

API

Application Program Interface.

architecture

Organizational structure of a system or component.

architectural design

Collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.

array

Collection of sensors distributed over a finite area (usually in a cross or concentric pattern) and referred to as a single station.

ASCII

American Standard Code for Information Interchange. Standard, unformatted 256-character set of letters and numbers.

authentication signature

Series of bytes that are unique to a set of data and that are used to verify the authenticity of the data.

authenticate

Verify the authenticity of a string of bits with an authentication signature.

B

bps

Bits per second. The speed at which data is transferred. Variants include Kbps and Mbps.

C

channel

Component of motion or distinct stream of data.

child process

UNIX process created by a parent process.

CMR

Center for Monitoring Research.

▼ Glossary

command

Expression that can be input to a computer system to initiate an action or affect the execution of a computer program.

component

(1) One dimension of a three-dimensional signal; (2) The vertically or horizontally oriented (north or east) sensor of a station used to measure the dimension; (3) One of the parts of a system; also referred to as a module or unit.

Comprehensive Nuclear-Test-Ban Treaty Organization

Treaty User group that consists of the Conference of States Parties (CSP), the Executive Council, and the Technical Secretariat.

Computer Software Component

Functionally or logically distinct part of a computer software configuration item, typically an aggregate of two or more software units.

Computer Software Configuration Item

Aggregation of software that is designated for configuration management and treated as a single entity in the configuration management process.

Conference of States Parties

Principal body of the CTBTO consisting of one representative from each State Party accompanied by alternate representatives and advisers. The CSP is responsible for implementing, executing, and verifying compliance with the Treaty.

configuration

Arrangement of computer system or component as defined by the number, nature, and interconnection of its parts.

configuration item

Aggregation of hardware, software, or both treated as a single entity in the configuration management process.

connection

Open communication path between protocol peers.

control flow

Sequence in which operations are performed during the execution of a computer program.

COTS

Commercial-Off-the-Shelf. Terminology that designates products such as hardware or software that can be acquired from existing inventory and used without modification.

cron

UNIX system utility to execute commands at regularly scheduled dates and times.

CSC

Computer Software Component.

CSCI

Computer Software Configuration Item.

CSP

I Conference of States Parties. The principal body of the CTBTO consisting of one representative from each State Party accompanied by alternate representatives and advisers. The CSP is responsible for implementing, executing, and verifying compliance with the Treaty.

CSS

Center for Seismic Studies (now the CMR).

CSS 3.0

Center for Seismic Studies (CSS) version 3 database schema, including a format for storing time-series data in disk files and database descriptors of that data.

CTBT

Comprehensive Nuclear-Test-Ban Treaty (the Treaty).

CTBTO

I Comprehensive Nuclear-Test-Ban Treaty Organization. Treaty user group that consists of the Conference of States Parties (CSP), the Executive Council, and the Technical Secretariat.

D**daemon**

Executable program that runs continuously without operator intervention. Usually, the system starts daemons during initialization. (Example: cron.)

data center

Location receiving data from multiple data providers. Often this location will also process and forward time-series data received.

Data Center Manager

Process to monitor and start selected components of the *CDS CD-1.1* Subsystem.

data consumer

Receiver of CD-1.1 Data Frames. This is always a data center.

data flow

Sequence in which data are transferred, used, and transformed during the execution of a computer program.

data provider

Sender of CD-1.1 Data Frames. This may be a station or a data center that forwards data.

DBA

Database Administrator.

DBMS

Database Management System.

detailed design

Refined and expanded version of the preliminary design of a system or component. This design is complete enough to be implemented.

▼ Glossary

disk loop

Storage device that continuously stores new waveform data while simultaneously deleting the oldest data on the device.

DSA

Digital Signature Algorithm.

E**entity-relationship (E-R) diagram**

Diagram that depicts a set of entities and the logical relationships among them.

execute

Carry out an instruction, process, or computer program.

F**failure**

Inability of a system or component to perform its required functions within specified performance requirements.

FIFO

First In First Out.

fork

UNIX system routine used by a parent process to create a child process that is an exact copy of itself.

frame

Logical collection of digital information that is transmitted as a unit from application to application.

frame handle

Data set describing a frame in the Frame Store.

frame set

Element of a Frame Store. Frame sets are defined for each source and destination pair. Each frame set has an associated frame log, which is also an element of the Frame Store.

Frame Store

Disk directories and files used to store raw CD-1.1 protocol frames. A Frame Store is made up of frame sets.

FTP

File Transfer Protocol. Protocol for transferring files between computers.

G**GB**

Gigabyte. A measure of computer memory or disk space that is equal to 1,024 megabytes.

GDI

Generic Database Interface.

H**HBO**

Host byte order.

host

Machine on a network that provides a service or information to other computers. Every networked computer has a host name by which it is known on the network.

Hz

Hertz.

I**IDC**

International Data Centre.

IGMP

Internet Group Management Protocol.

IMS

International Monitoring System.

inetd

Internet services daemon for UNIX. The daemon listens for service requests on a TCP or UDP port and executes the server program associated with the service.

instance

Running computer program. An individual program may have multiple instances on one or more host computers.

Internet

World-wide network of computers linked by means of the IP protocol.

I/O

Input/Output.

IP

Internet Protocol.

IP address

Internet Protocol address, for example: 140.162.1.27.

IPC

Interprocess communication. The messaging system by which applications communicate with each other through *libipc* common library functions.

J**job template**

A series of parameter definitions used to describe a job/process to be managed by the [Data Center Manager](#).

K**key**

Data string used by authentication software. Typically keys are defined in pairs, public and private. The private key is used to sign data (produce a validation data value), and the public key is used to verify data (determine that a validation data value was produced by the private counterpart of the public key).

L**LAN**

Local Area Network.

▼ **Glossary**

LIFO

Last In First Out.

M

M

Mega (prefix), million.

MB

Megabyte. 1,024 kilobytes.

MHz

Megahertz. A million cycles (occurrences, alterations, pulses) per second.

monitoring system

See [IMS](#).

MTU

Maximum Transmission Unit. The largest packet size that a network can transmit.

multicast

To transmit a single message to a select group of recipients over a network.

mutex

Mutual exclusion lock used to prevent multiple threads from simultaneously executing critical sections of code that access shared data.

N

NBO

Network byte order.

NDC

National Data Center.

network

Two or more computer systems linked together.

NFS

Network File System (Sun Microsystems). Protocol that enables clients to mount remote directories onto their own local filesystems.

O

ORACLE

Vendor of PIDC and IDC database management system.

P

par

See [parameter](#).

parameter

User-specified token that controls some aspect of an application (for example, database name, threshold value). Most parameters are specified using [*token* = *value*] strings, for example, `dbname=mydata/base@oracle`.

parameter (par) file

ASCII file containing values for parameters of a program. Par files are used to replace command line arguments. The files are formatted as a list of [*token* = *value*] strings.

parent process

UNIX process that creates a child process.

parse

Decompose information contained in a set of data.

pathname

Filesystem specification for a file's location.

PID

Process Identifier.

PIDC

Prototype International Data Centre.

pipe

Interprocess communication facility provided by the UNIX operating system. Pipes typically are defined in pairs to support data transmission between two processes where each pipe supports a one-way flow of data.

PNack

Packet Negative Acknowledgement.

port

Connection to a computer.

protocol

Common set of rules and signals that computers on a network use to communicate.

protocol peer

Computer system participating in an exchange of data using a specific protocol (for example CD-1.1).

R**RAM**

Random Access Memory.

recovery

Restoration of a system, program, database, or other system resource to a state in which it can perform required functions.

run

(1) Single, usually continuous, execution of a computer program. (2) To execute a computer program.

S**SAIC**

Science Applications International Corporation.

schema

Database structure description.

script

Small executable program, written with UNIX and other related commands, that does not need to be compiled.

signature

Bits added to data that are used to verify the authenticity of the data.

▼ Glossary

socket

Type of file used for network communication between processes.

software unit

Discrete set of software statements that implements a function; usually a sub-component of a CSC.

Solaris

Name of the operating system used on Sun Microsystems hardware.

spawn

To launch a program from another program. The child program is spawned from the parent program.

SSL

Secure Sockets Layer. A protocol developed by Netscape for transmitting private documents via the Internet. SSL works by using a public key to encrypt data that's transferred over the SSL connection.

States Parties

Treaty user group who will operate their own or cooperative facilities, which may be NDCs.

T**TCP/IP**

Transmission Control Protocol/Internet Protocol.

thread

Short for a processing thread, which is an execution path. A process may be single or multi-threaded. In a multi-threaded process all threads share a single address space but have independent execution paths.

Treaty

Comprehensive Nuclear-Test-Ban Treaty (CTBT).

U**UDP**

User Datagram Protocol.

unicast

Communication that takes place over a network between a single sender and a single receiver.

UNIX

Trade name of the operating system used by the Sun workstations.

W**WAN**

Wide Area Network.

waveform

Time-domain signal data from a sensor (the voltage output) where the voltage has been converted to a digital count (which is monotonic with the amplitude of the stimulus to which the sensor responds).

Web

- | World Wide Web. A graphics-intensive environment running on top of the Internet.

wfdisc

Waveform description record or table.

Index

A

affiliation [28](#), [119](#), [137](#), [138](#)

Alpha protocol [12](#)

alphasite [28](#), [59](#), [137](#), [138](#)

Authentication

functional description [29](#), [41](#)

Authentication Signing

requirements [149](#)

requirements traceability [166](#)

C

CDS CD-1.1

data flow model [46](#)

functional description [29](#)

functional requirements [142](#)

interface design [42](#)

system requirements [153](#)

channame [138](#)

configuration

hardware [8](#)

connection initiation

multicast [26](#)

unicast catchup [26](#)

Connection Manager

context [54](#)

control [60](#)

database processing [58](#)

data flow model [46](#), [50](#)

detailed design [54](#)

error states [60](#)

frame processing [58](#)

functional description [29](#), [34](#)

I/O [54](#)

interfaces [60](#)

output [59](#)

processing [55](#)

requirements [142](#)

requirements traceability [157](#)

Connection Manager Server

components [64](#)

context [62](#)

control [66](#)

data flow model [46](#), [50](#)

detailed design [61](#)

error states [66](#)

I/O [62](#)

interfaces [66](#)

Connection Originator

context [75](#)

control [77](#)

data flow model [49](#)

detailed design [75](#)

error states [78](#)

functional description [29](#), [36](#)

I/O [77](#)

interfaces [77](#)

internal data and control flow [76](#)

requirements [144](#)

requirements traceability [159](#)

Connection Response Frame

contents [26](#)

COTS

openssl [9](#)

ORACLE DBMS [9](#)

requirements [9](#)

cron

▼ Index

running *Frame Store Stager* [41](#)

D

database

Database Management System (DBMS)

DBMS [8](#)

description [136](#)

interface [136](#)

schema overview [28](#)

table relationships [138](#)

usage by CDS [139](#)

use of [9](#)

use of and access to [18](#)

Data Center Manager

components [68](#)

context [67](#)

control [72](#)

data flow model [48](#), [51](#)

detailed design [67](#)

error states [74](#)

events [71](#)

functional description [29](#), [35](#)

I/O [69](#)

interfaces [72](#)

internal control flow [74](#)

job template attributes [70](#)

requirements [143](#)

requirements traceability [158](#)

data flow symbols [vi](#)

data packet

format [135](#)

Data Parser

context [114](#)

control [120](#)

data flow [115](#)

data flow model [48](#)

detailed design [113](#)

DLParse Exec [115](#)

error states [121](#)

functional description [29](#), [40](#)

I/O [119](#)

interfaces [121](#)

output [120](#)

Process Frame [119](#)

Process Loop [117](#)

requirements [147](#)

requirements traceability [163](#), [165](#)

design model

multicast operation [21](#)

traditional unicast operation [20](#)

development

history [7](#)

dlfile [119](#), [137](#), [138](#)

dlman [28](#), [58](#), [137](#), [138](#)

E

entity-relationships [138](#)

entity-relationship symbols [vii](#)

Exchange Controller

context [79](#)

control [84](#)

Controller Executive [81](#)

data flow [80](#)

data flow model [47](#), [50](#)

detailed design [79](#)

error states [87](#)

Exchange Interface [82](#)

Frame Handler [83](#)

functional description [29](#), [37](#)

I/O [84](#)

interfaces [85](#)

requirements [145](#)

requirements traceability [160](#)

F

file system

use of [19](#)

Frame Exchange

components [89](#)

context [88](#)

- control [95](#)
- data flow model [48](#), [50](#)
- detailed design [88](#)
- error states [97](#)
- Frame I/O* [92](#)
- functional description [29](#), [38](#)
- Heartbeat* [91](#)
- I/O [94](#)
- interfaces [96](#)
- Main Loop* [90](#)
- Message Sender* [91](#)
- requirements [146](#)
- requirements traceability [161](#)
- Sender* [92](#)
- Time Counter* [90](#)
- Frame Store
 - requirements [151](#)
 - requirements traceability [168](#)
- Frame Store Stager*
 - functional description [29](#), [41](#)
- functional requirements [142](#)

H

- hardware
 - configuration [8](#)
 - requirements [7](#)

I

- IGMP [9](#)
- inetd
 - running *Connection Manager* [46](#)
- instrument** [119](#), [137](#), [138](#)
- interfaces
 - database [136](#)
 - external users [43](#)
 - operator [43](#)
 - other IDC systems [42](#)
- IPC
 - use of [18](#)

L

- libas* [17](#)
- libcancomp* [17](#)
- libcdo*
 - control [134](#)
 - detailed design [133](#)
 - error states [135](#)
 - I/O [134](#)
 - requirements [152](#)
 - requirements traceability [170](#)
- libframelog* [17](#)
- libfs*
 - control [131](#)
 - detailed design [128](#)
 - error states [133](#)
 - I/O [129](#)
 - interfaces [131](#)
- libgdi* [17](#)
- liblog* [17](#)
- libpar* [17](#)
- libraries
 - used by *CDS CD-1.1* [17](#)
- libstdtime* [17](#)
- libtable* [17](#)
- libwfm* [17](#)
- libwio* [18](#)
- log files [43](#)

M

- Missing Frame Detector* [109](#)
 - context [109](#)
 - control [112](#)
 - data flow [110](#)
 - data flow model [50](#)
 - error states [112](#)
 - I/O [111](#)
 - interfaces [112](#)
 - processing [109](#)
- multicast
 - data packet [135](#)

▼ Index

- PNack packet [136](#)
- protocol [135](#)
- reliable multicasting [21](#)
- transmission [13](#)
- multicast group
 - IGMP [9](#)
 - joining [26](#)
 - membership [9](#)
- multicast operation
 - conceptual design [15](#)
 - connection initiation
 - multicast [26](#)
 - unicast catchup [26](#)
 - data flow model [49](#)
 - design model [21](#)
 - functional description [32](#)
 - startup time [27](#)
- Multicast Receiver*
 - data flow model [49](#)
 - Error States [108](#)
 - functional description [29](#), [39](#)
 - Processing [104](#)
- Multicast Sender*
 - context [98](#)
 - Error States [103](#)
 - functional description [29](#), [38](#)
 - Processing [99](#)
- multicast subsystem
 - data flow [51](#)

N

- network requirements [8](#)

O

- openssl* [18](#)
 - download [9](#)
 - use of [9](#)
- operating environment [7](#)
- operator interface [43](#)

P

- packet format
 - data packet [135](#)
 - PNack packet [136](#)
- PNack packet
 - format [136](#)
 - transmission [21](#)
- protocol
 - CD-1.0 (Alpha) [12](#)
 - CD-1.1 [12](#)
 - multicast [135](#)
- Protocol Checker*
 - context [126](#)
 - control [127](#)
 - detailed design [126](#)
 - error states [128](#)
 - I/O [127](#)
 - interfaces [127](#)

R

- reliability hosts [24](#)
- reliable multicasting [21](#)
- requirements
 - Authentication Signing* [149](#)
 - Connection Manager* [142](#)
 - Connection Originator* [144](#)
 - COTS software [9](#)
 - Data Center Manager* [143](#)
 - Data Parser* [147](#)
 - Exchange Controller* [145](#)
 - Frame Exchange* [146](#)
 - Frame Store* [151](#)
 - functional [142](#)
 - hardware [7](#)
 - libcdo* [152](#)
 - network [8](#)
 - Signature Authentication* [150](#)
 - system [153](#)

S

sensor [28](#), [119](#), [137](#), [138](#)

Signature Authentication

requirements [150](#)

requirements traceability [167](#)

site [28](#), [119](#), [137](#), [138](#)

sitechan [28](#), [119](#), [137](#), [138](#)

software requirements

COTS [9](#)

system requirements

requirements traceability [171](#)

disk loop [28](#)

files [17](#)

wfconv [29](#), [119](#), [137](#), [138](#)

wfdisc [29](#), [40](#), [49](#), [119](#), [137](#)

wfproto [29](#), [119](#), [137](#)

T

TCP/IP [8](#), [43](#)

typographical conventions [vii](#)

U

UDP [8](#), [43](#)

unicast

traditional unicast [14](#)

transmission [13](#)

UDP [21](#)

unicast catchup [22](#)

unicast catchup subsystem

data flow [51](#)

reliability hosts [24](#)

unicast operation, traditional

conceptual design [14](#)

data flow model [46](#)

design model [20](#)

functional description [30](#)

W

waveform

CSS 3.0 format [49](#), [136](#)

DBMS description records [17](#)

